Information Sciences 399 (2017) 98–120

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

CrowdK: Answering top-*k* queries with crowdsourcing

Jongwuk Lee[a,*], Dongwon Lee[b], Seung-won Hwang[c]

[a] Department of Software, Sungkyunkwan University, Republic of Korea
[b] College of Information Sciences and Technology, The Pennsylvania State University, PA, USA
[c] Department of Computer Science, Yonsei University, Republic of Korea

ARTICLE INFO

Article history:
Received 29 January 2016
Revised 3 March 2017
Accepted 4 March 2017
Available online 6 March 2017

Keywords:
Crowdsourcing
Top-*k* queries
Parameterized framework
Dynamic programming

ABSTRACT

In recent years, crowdsourcing has emerged as a new computing paradigm for bridging the gap between human- and machine-based computation. As one of the core operations in data retrieval, we study top-*k* queries with crowdsourcing, namely *crowd-enabled top-k queries*. This problem is formulated with three key factors, *latency*, *monetary cost*, and *quality of answers*. We first aim to design a novel framework that minimizes monetary cost when latency is constrained. Toward this goal, we employ a two-phase *parameterized framework* with two parameters, called *buckets* and *ranges*. On top of this framework, we develop three methods: *greedy*, *equi-sized*, and *dynamic programming*, to determine the buckets and ranges. By combining the three methods at each phase, we propose four algorithms: GdyBucket, EquiBucket, EquiRange, and CrowdK. When the crowd answers are imprecise, we also address improving the accuracy of the top-*k* answers. Lastly, using both simulated crowds and real crowds at Amazon Mechanical Turk, we evaluate the trade-off between our proposals with respect to monetary cost, accuracy of answers, and running time. Compared to other competitive algorithms, it is found that CrowdK reduces monetary cost up to 20 times, without sacrificing the accuracy of the top-*k* answers.

1. Introduction

Crowdsourcing has emerged as a new computing paradigm for human computation and has been widely used for bridging the gap between machine- and human-based computation. Humans can realize considerably improved results compared to computers when performing intelligent tasks such as answering users' semantic search queries [6], understanding topics in microblogs [19], image tagging for subjective topics [43], editing a document as natural language understanding [4], and choosing a representative frame that summarizes a video [3]. Furthermore, crowdsourcing can also be used to collect labels for machine learning [31] and to perform tasks in human-computer interaction [9,36].

Databases are another field where crowdsourcing can contribute. Well-known crowd-enabled databases include Deco [33], CrowdDB [16], Qurk [28], CDAS [26], CyLog/Crowd4U [30], and AskIt! [5]. At a micro level, existing work has extended machine-based operations into crowd-enabled operators, *e.g.*, selection [17,32,37], join [29,41,42], group by [12], sorting [29], max [21,38,39], and skyline queries [25,27]. It is particularly effective for addressing semantic and subjective concepts in query operations.

* Corresponding author.
E-mail addresses: jongwuklee@skku.edu (J. Lee), dongwon@psu.edu (D. Lee), seungwonh@yonsei.ac.kr (S.-w. Hwang).

http://dx.doi.org/10.1016/j.ins.2017.03.010
0020-0255/

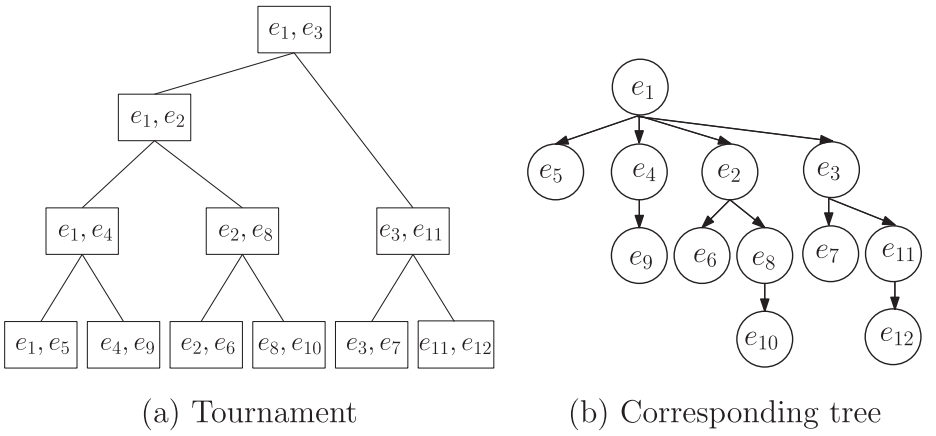


Fig. 1. Building tournament, taking 11 questions to find top-1 item.

As one of core the operations in data retrieval, this paper studies top-k queries with crowdsourcing, namely *crowd-enabled top-*k *queries*. By extending the traditional exact match query model in databases, we retrieve a ranked list of the seemingly-best k items. In this paper, our main question is: "*How do we leverage the power of crowdsourcing to find meaningful answers for top-*k *queries?*" To illustrate this, we describe the following example.

Example 1 (Crowd-enabled top-k query)**.** A user may formulate a top-k query to find the most relevant photos of "New York City" from 2000 to 2015:

```
SELECT * FROM photo_db
WHERE location = 'New York City'
      and year >= 2000 and year <= 2015
ORDER BY relevance LIMIT 5
```

To achieve high accuracy in machine-based systems, the user should define an appropriate notion of "relevance." However, it is not trivial to specify the scoring function for relevance to "New York City". In clear contrast, one can ask a large number of human workers what photos are relevant to "New York City". Using the crowd, we can retrieve the most meaningful top-5 photos without specifying a complicated scoring function. In particular, when the crowd-enabled query is executed on mobile devices (*e.g.*, [43]) or requires fast query processing time (*e.g.*, [41]), it is important to minimize the cost for crowds under a constraint for response time.

In practice, we envision that crowd-enabled top-k queries are likely to be used in a hybrid fashion. That is, machine-based computation is first applied for hard Boolean constraints to remove unnecessary items (*e.g.*, finding photos pictured in 2000–2015) and then crowdsourcing can be used to prioritize the remaining items (*e.g.*, finding the most relevant top-5 photos). In this sense, the number of items used in crowdsourcing can significantly reduced, *e.g.*, less than 10,000 items.

To address crowd-enabled top-k queries, we consider three key factors used in crowdsourcing systems: (1) *latency*: how to control the inherent delay of collecting answers from crowds (or workers), (2) *monetary cost*: how much to pay crowds, and (3) *quality of answers*: how to manage erroneous answers returned by crowds. Existing work has developed various crowd-enabled algorithms with the three factors. Venetis et al. [38] studied *max* queries for optimizing the accuracy of the top-1 answer when both monetary cost and latency were constrained. Davidson et al. [12] addressed the problem of crowd-enabled top-k queries without considering a latency constraint. Without considering the constraint for latency, Polychronopoulos et al. [35] focused on retrieving a *top-*k *set*, where the ordering of top-k items was ignored.

In this paper, we aim to develop a crowd-enabled top-k algorithm with the three factors. (In Section 6, we elaborate on the differences between the proposed algorithms and existing work.) First, assuming that the answers of workers are always precise, we propose a crowd-enabled top-k algorithm that minimizes monetary cost, when latency is constrained. For simplicity, we exchangeably use the task and the question asked to the crowd. This scenario is particularly effective for supporting real-time query computation [2,22]. As the simple model for latency, existing work [21,38,39] assumes that workers return their answers within a specific response time, *e.g.*, 30 min. This time range is referred to as a *round*. Because multiple questions can be asked to crowds within the response time, they can be viewed as parallelizing questions in the round. Note that, because a crowd-enabled algorithm aims to minimize the number of questions under the given rounds, it is clearly different from traditional optimization that maximizes CPU efficiency [24].

To minimize the number of questions, we employ a *parameterized framework* with two parameters, inspired by *tournament sort* (Fig. 1). *Buckets* and *ranges* are used in the following phases.

Table 1
List of notations.

Notation	Definition
$\mathcal{E}$	Set of n items, *i.e.*, $\{e_1, \ldots, e_n\}$
n	Number of items in $\mathcal{E}$, *i.e.*, $n = \|\mathcal{E}\|$
k	Retrieval size, $k \leq n$
$\pi(e_i)$	Latent scoring function
$\prec\succ$	Indifference on $\mathcal{E}$
$\succ$	Precedence on $\mathcal{E}$
$\simeq$	Equivalence on $\mathcal{E}$
A	Execution plan for top-k queries
τ	Number of rounds
γ	Monetary cost per question
ω	Number of workers per question
$\mathcal{Q}$	Total set of questions, $\{\mathcal{Q}_1, \ldots, \mathcal{Q}_\tau\}$
$\mathcal{Q}_i$	Set of questions at ith round
$\mathcal{B}$	Total sequence of bucket sizes, $\langle \mathcal{B}_1, \ldots, \mathcal{B}_{\tau_\mathcal{B}} \rangle$
$\mathcal{R}$	Sequence of ranges, $\langle r_0, \ldots, r_{\tau_U} \rangle$
$\mathcal{B}_i$	Sequence of bucket sizes at ith round
$\|\mathcal{B}_i\|$	Number of buckets for $\mathcal{B}_i$
b_{ij}	jth bucket size in $\mathcal{B}_i$
$\widetilde{\mathcal{B}}$	Sequence of vertical bucket sizes
$\mathcal{T}$	Transformed tree for A

1. *Building a tree:* We partition an item set into disjoint subsets of items, called buckets. At each bucket, the questions for items are asked to crowds, and the precedence of items is determined. For the sake of representation, the items and precedences between items can be represented by nodes and directed edges, respectively. By iteratively asking the precedences between items at each round, a *tree* is built, where the root implies the top-1 item.
2. *Updating a tree:* After building a tree, a set of retrieval sizes, called a range, is used to identify the top-k answers. The retrieval size is the number of best items requested by the user, *e.g.*, retrieving the top-5 items in Example 1. Given a range, the questions for the top-k candidates are asked and the tree is updated incrementally. We can also avoid asking unnecessary questions using the pre-obtained precedences between items.

Because the number of questions highly depends on buckets and ranges under the latency constraint, our goal is to determine the *optimal* values for buckets and ranges that minimize the number of questions. To determine buckets and ranges, we develop three methods on top of the parameterized framework: *greedy, equi-sized*, and *dynamic programming*. We then propose four algorithms: GdyBucket, EquiBucket, EquiRange, and CrowdK by combining the three methods at each phase.

We then address the scenario where the crowd answers are imprecise. There are two approaches for improving the accuracy of the top-k answers. As a query-independent approach, it is possible to improve the accuracy per question as in [17,26,32]. As a query-dependent approach, when computing crowd-enabled top-k queries, specific questions can be more important for achieving accurate top-k answers. That is, we can assign a different number of workers per question depending on query execution plans. We thus develop how to distinguish the *importance* of questions.

To summarize, the main contributions of this paper are as follows:

- We formulate the problem of crowd-enabled top-k queries (Section 2) and model a two-phase framework with two parameters (Section 3).
- We develop three methods on top of the parameterized framework: *greedy, equi-sized*, and *dynamic programming* (Section 4).
- We propose four algorithms: GdyBucket, EquiBucket, EquiRange, and CrowdK, by combining the three methods (Section 4).
- We discuss how to improve the accuracy of answers using static and dynamic worker assignment methods (Section 4).
- We validate the trade-off between the proposed algorithms using both a simulated crowdsourcing and a real environment at Amazon Mechanical Turk (Section 5).

2. Preliminaries

In this section, we first introduce the basic notation used to formulate the crowd-enabled top-k query problem. Let $\mathcal{E} = \{e_1, \ldots, e_n\}$ denote a finite set of n items with no duplicates. When the scoring function of the top-k query is unspecified, we adopt crowdsourcing to assess a latent scoring function $\pi(e_i)$ of item e_i. If the precedence between e_i and e_j is not yet determined, e_i and e_j are said to be *indifferent*, denoted by $e_i \prec\succ e_j$. If $\pi(e_i) > \pi(e_j)$, e_i is said to be *better* than e_j, denoted by $e_i \succ e_j$. If $\pi(e_i) = \pi(e_j)$, e_i and e_j are *equivalent*, denoted by $e_i \simeq e_j$. Further, if $\pi(e_i) \geq \pi(e_j)$, it is denoted by $e_i \succeq e_j$. For $\forall e_i, e_j, e_k \in \mathcal{E}$, if $e_i \succeq e_j$ and $e_j \succeq e_k$, then $e_i \succeq e_k$ holds by *transitivity*. Table 1 summarizes the notations used in this paper.

2.1. Problem definition

Given an item set $\mathcal{E}$ and a retrieval size k ($\leq n$), the crowd-enabled top-k query returns an *ordered set* of k items $\mathcal{K} = \{e_1, \ldots, e_k\}$ such that $e_1 \succeq \ldots \succeq e_k$ and $\forall e_i \in \mathcal{K}, e_j \in \mathcal{E} \backslash \mathcal{K} : e_i \succeq e_j$. We stress that crowds determine all precedences between items for the latent scoring function. The latent scores of items are thus consistently qualified by the crowd. (In a real scenario, the answers of the crowd can conflict. In that case, we borrow the voting method in [21] to resolve the conflict.)

The first step in crowd-enabled queries is to design the format of the micro-tasks. Depending on the characteristics of the crowd-enabled queries, we can create different formats, *e.g.*, true-false, binary-choice, or multiple-choice. As a simple format, in this paper, we adopt a *pair-wise question* (e_i, e_j) with a binary answer, as widely used in existing work [12,21,35,38,39]. Given (e_i, e_j), crowds (or *workers*) are asked to choose the better one of the two items. When the crowd determine that the two items are equivalent, it is broken arbitrarily. For the sake of representation, we interchangeably use both micro-tasks and *questions* asked to the crowd.

Before performing crowd-enabled queries, it is also necessary to build a schedule for asking questions to crowds under the given constraint. We call this *execution plan A*. This can be used to estimate the total monetary cost, as discussed in [38]. Specifically, we address the following three factors in designing the execution plan.

(1) *Latency:* When A is created for $\mathcal{E}$, the latency can be measured by the response time required to run A, denoted by $T(A, \mathcal{E})$. Based on the notion of rounds (or *iterations*) [21,38,39], the questions can be asked to crowds in parallel.[1] $T(A, \mathcal{E})$ can be measured by the number τ of rounds. We can consider two strategies [21] depending on the values of τ: (i) *one-shot strategy* ($\tau = 1$) submits all possible questions at one time, and (ii) *adaptive strategy* ($\tau \geq 2$) submits initial questions, receives their answers, submits more questions, continuing. The one-shot strategy can be faster than the adaptive strategy; however it can incur all possible questions in the worst case. (Using the pair-wise question, $\binom{n}{2}$ can be asked to crowds in the worst case.) The adaptive strategy requires more rounds than the one-shot strategy. However, the questions are asked selectively using the pre-obtained precedences of the items at the previous rounds.

(2) *Monetary cost:* When questions are asked to crowds, we should pay a certain reward. When A is performed for $\mathcal{E}$, let $C(A, \mathcal{E})$ denote an estimated total monetary cost of A. When other factors are constrained, it is critical to minimize the monetary cost. (We will later explain how to calculate $C(A, \mathcal{E})$ for a given execution plan.)

(3) *Quality of answers:* To quantify the accuracy of crowd-enabled queries, denoted by $Q(A, \mathcal{E})$, existing metrics, *e.g.*, precision and recall, can be used. When crowds always return precise answers, $Q(A, \mathcal{E})$ is 1.0. In reality, because some answers can be erroneous, $Q(A, \mathcal{E})$ can be less than 1.0. To address this problem, existing work [17,26,32] developed various models by assigning multiple workers per question. Let ω denote the number of workers to be assigned per question. For simplicity, ω workers are equally allocated per question and ω answers can be aggregated using *majority voting*.

Assuming that the answers to the questions are always correct, we first aim to minimize the monetary cost when latency is constrained. This scenario is particularly useful for supporting real-time environments [2,22]. Formally, it is formulated as follows:

Definition 1 (Crowd-enabled top-k query)**.** Given a set of items $\mathcal{E}$, a retrieval size k, and a round bound τ, we aim to identify a *cost-optimal execution plan* A^{opt} that returns the top-k answers such that:

$$A^{opt} = \underset{A \in \mathcal{A}}{\operatorname{argmin}} C(A, \mathcal{E}) \quad \text{subject to} \quad T(A, \mathcal{E}) \leq \tau.$$

We build an execution plan A that consists of a sequence of question sets $\mathcal{Q} = \{\mathcal{Q}_1, \ldots, \mathcal{Q}_\tau\}$ through τ rounds such that $T(A, \mathcal{E}) \leq \tau$. Let $\mathcal{Q}_i$ be a set of questions at the ith round, and $|\mathcal{Q}_i|$ be the number of questions for $\mathcal{Q}_i$. Let γ denote a fixed cost per question, *e.g.*, \$ 0.05. $C(A, \mathcal{E})$ can thus be computed:

$$C(A, \mathcal{E}) = \gamma \cdot \omega \cdot \sum_{i=1}^{\tau} |\mathcal{Q}_i|. \tag{1}$$

Assuming that the answers to the questions are always correct, ω is equally set as 1. (In Section 4.4 we will relax this assumption and discuss a method for assigning a different number of workers per question.) Therefore, it is essential to minimize the total number $|\mathcal{Q}|$ of questions in $C(A, \mathcal{E})$.

2.2. Tournament sort

As a baseline method, we adopt *tournament sort* [11], which can be easily extended for crowd-enabled settings. Basically, pair-wise comparisons are replaced by the pair-wise questions asked to the crowd. When executing the tournament sort, the precedence between items can be visualized by a *tree* $\mathcal{T}$. That is, nodes are items and directed edges are the precedences between the items. Based on the tree, we explain how to build and update the tree for identifying the top-k answers.

[1] The latency can be modeled as the response time of workers. For simplicity, in this paper, the latency is measured by the number of rounds.

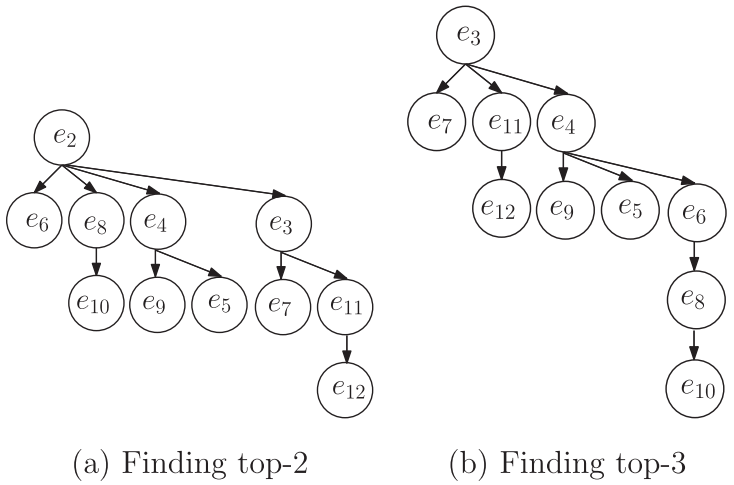


Fig. 2. Updating tournament trees to retrieve top-2 and top-3 items.

Table 2
Analysis of tournament sort.

	$k = 1$	$k \geq 2$
# of questions	$O(n)$	$O(n + k\lceil \log_2 n \rceil)$
# of rounds	$O(\lceil \log_2 n \rceil)$	$O(k\lceil \log_2 n \rceil)$

(1) *Building a tree:* An item set $\mathcal{E}$ is partitioned into $\lceil n/2 \rceil$ subsets. At the bottom level, two consecutive items are compared. In this process, multiple questions are asked together in this one round. Then, the better item (called the *winner*) is promoted to the upper level. This process iterates at each level until one item remains. This process yields $n - 1$ questions during $\lceil \log_2 n \rceil$ rounds. After building a tree, the root node is the top-1 item.

(2) *Updating a tree:* To determine the top-k answers ($k \geq 2$), the tree is incrementally updated. When the top-1 item is removed from the tree, the top-2 candidates are considered the items that were compared with and lost to the top-1 item. That is, the height $\lceil \log_2 n \rceil$ of the tree implies the number of top-2 candidates. Among top-2 candidates, the two items at the bottom level are compared, and the winner is compared with another item at the upper level. Because the questions have a dependency, a single question is performed for each round. Therefore, $\lceil \log_2 n \rceil - 1$ questions are asked during $\lceil \log_2 n \rceil - 1$ in the worst case.[2] The updating process is repeated $k - 1$ times to identify the top-k answers.

Example 2 (Tournament sort)**.** Suppose that we have a set of 12 items and retrieval size $k = 3$. For brevity, we assume that an item with a smaller script number is preferred, *e.g.*, $e_1 \succ e_2$, and items are randomly distributed. Fig. 1(a) depicts a tournament for a given toy dataset. At the first round, six questions $\{(e_1, e_5), (e_4, e_9), (e_2, e_6), (e_8, e_{10}), (e_3, e_7), (e_{11}, e_{12})\}$ are asked. The winners $\{e_1, e_4, e_2, e_8, e_3, e_{11}\}$ are promoted to the upper level, and three questions $\{(e_1, e_4), (e_2, e_8), (e_3, e_{11})\}$ are asked at the next round. Similarly, $\{(e_1, e_2)\}$ and $\{(e_1, e_3)\}$ are asked serially. In this process, 11 questions are generated during four rounds, and e_1 is identified as the top-1 item. Fig. 1(b) depicts a tree that corresponds to the tournament in Fig. 1(a).

To identify the top-2 item (second item), the tree is updated incrementally. The root item e_1 is removed from the tree; $\{e_5, e_4, e_2, e_3\}$ are the top-2 candidates that lost to e_1. For these items, three questions, (e_5, e_4), (e_4, e_2), and (e_2, e_3) are asked serially. (For the items connected to item e_1, the questions are generated from the left to right direction.) At the end, e_2 is determined as the top-2 item. In this process, because three questions are asked serially, three rounds are required. Fig. 2(a) describes the updated tree after identifying the top-2 item.

Similarly, $\{e_6, e_8, e_4, e_3\}$ are top-3 candidates that have lost to e_2. After asking three questions (e_6, e_8), (e_6, e_4), and (e_4, e_3) during three rounds, e_3 is determined as the top-3 item as shown in Fig. 2(b). In summary, $\{e_1, e_2, e_3\}$ are identified as the top-3 answers, yielding 17 questions over ten rounds.

Table 2 presents the informal analysis of the tournament sort in terms of the number of questions and number of rounds. If the number of rounds is not constrained, the tournament sort can be used to build an execution plan that minimizes the number of questions. (This is because the tournament sort is optimal in terms of the number of pair-wise comparisons.)

[2] Although it is possible to ask some questions together, *e.g.*, (e_5, e_4) and (e_2, e_3) in Fig. 1(b), traditional tournament sort does not handle the optimization for latency.

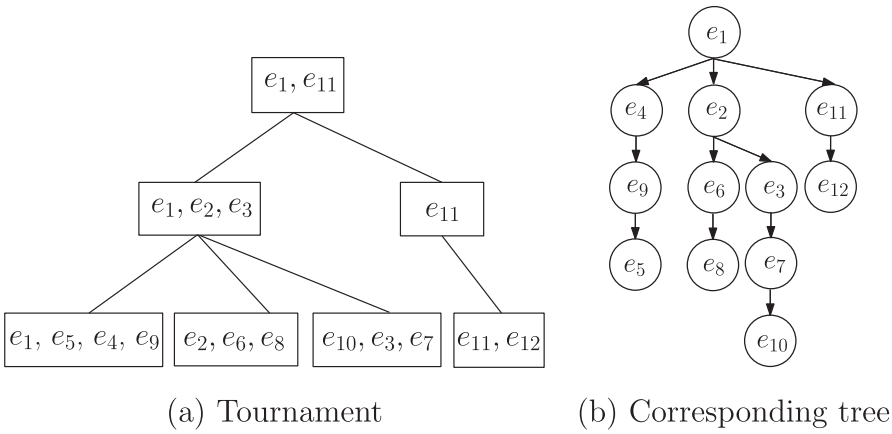


Fig. 3. Tournament with different bucket sizes.

When the number of rounds is constrained, we should develop an alternative execution plan. Therefore, our goal is to design a generalized framework to develop an execution plan that minimizes the number of questions under an arbitrary latency constraint.

3. Parameterized framework

We first present two key principles used to design the proposed model for computing crowd-enabled top-k queries.

1. *Validity of top-*k *answers:* assuming that crowds always provide correct answers, execution plan A returns *valid* top-k answers with no wild guesses. That is, after plan A is executed, A should guarantee to determine an ordered set $\mathcal{K} = \{e_1, \ldots, e_k\}$ of top-k items such that $e_1 \succeq \cdots \succeq e_k$ and $\forall e_i \in \mathcal{K}$, $e_j \in \mathcal{E} \backslash \mathcal{K} : e_i \succeq e_j$.
2. *Minimality of questions:* assuming that no pre-defined preferences between items exist, all questions made by A should contribute to deciding the valid top-k answers under a given latency constraint. That is, all questions should be related to identify top-k items.

Based on these two principles, we model a *parameterized framework* $\mathcal{A}(\mathcal{B}, \mathcal{R})$ with two parameters, inspired by tournament sort (Section 2.2). Depending on the values of $\mathcal{B}$ and $\mathcal{R}$, different execution plans can be performed. Each parameter is involved in building and updating a tree as in tournament sort. The total cost $C(A, \mathcal{E})$ can thus be calculated by summing two costs: (i) the cost $C_B(A, \mathcal{E})$ of building a tree and (ii) the cost $C_U(A, \mathcal{E})$ of updating a tree. Let τ_B $(0 < \tau_B \leq \tau)$ and τ_U $(\tau_U = \tau - \tau_B)$ denote the number of rounds used in building and updating a tree, respectively.

Specifically, we explain how $\mathcal{B}$ and $\mathcal{R}$ are used in computing $C_B(A, \mathcal{E})$ and $C_U(A, \mathcal{E})$, respectively.

(1) *How to determine* $\mathcal{B}$*:* While executing tournament sort, two consecutive items are compared in parallel. In Fig. 1, an item set $\mathcal{E}$ is partitioned into six subsets at the bottom level, *i.e.*, $\{e_1, e_5\}$, $\{e_4, e_9\}$, $\{e_2, e_6\}$, $\{e_8, e_{10}\}$, $\{e_3, e_7\}$, and $\{e_{11}, e_{12}\}$. Each subset is called a *bucket*. If one wants to reduce the number of rounds, one may increase the size of buckets from two to three. That is, the size of the buckets can be different from the latency constraint.

Given τ_B, $\mathcal{B} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_{\tau_B} \rangle$ refers to a sequence of bucket sizes in building a tree. When partitioning an item set $\mathcal{E}$ into disjoint buckets, $\mathcal{B}_i = \langle b_{i1}, \ldots, b_{i|\mathcal{B}_i|} \rangle$ is a list of bucket sizes partitioned at the ith round. Let b_{ij} be the jth bucket size at the ith round. Given $b_{ij} \in \mathcal{B}_i$, all pair-wise questions $(\frac{b_{ij}}{2})$ are necessary to determine the valid top-k items. For example, given $\mathcal{B}_1 = \langle 4, 4, 2, 2 \rangle$, the items can be partitioned into $\{e_1, e_5, e_4, e_9\}$, $\{e_2, e_6, e_8, e_{10}\}$, $\{e_3, e_7\}$, and $\{e_{11}, e_{12}\}$ at the first round. In that case, 14 questions are asked (Fig. 4a). We can obtain an ordered list of items at each bucket, *i.e.*, $e_1 \succ e_4 \succ e_5 \succ e_9$, $e_2 \succ e_6 \succ e_8 \succ e_{10}$, $e_3 \succ e_7$ and $e_{11} \succ e_{12}$. For each subset, the winners $\{e_1, e_2, e_3, e_{11}\}$ are promoted to the next round.

(2) *How to determine* $\mathcal{R}$*:* The top-k answers are obtained by incrementally updating a tree. To satisfy a latency constraint, we can perform more questions when updating a tree. In Fig. 2, three questions (e_5, e_4), (e_4, e_2), and (e_2, e_3) are asked to the crowds during three rounds. However, given only one round, we should ask all pair-wise comparisons for $\{e_5, e_4, e_2, e_3\}$ to identify the top-2 item. Similarly, all pair-wise comparisons for $\{e_9, e_6, e_8, e_7, e_{11}\}$ are appended to retrieve the top-3 item over one round.

Given the number τ_U of rounds and a retrieval size k, $\mathcal{R} = \langle r_0, \ldots, r_{\tau_U} \rangle$ refers to a sequence of retrieval sizes such that $\forall r_i \in \mathcal{R} : r_i \leq k$ and $r_0 < r_1 \leq \ldots \leq r_{\tau_U}$. Let r_i denote the retrieval size at the ith round. As a special case, let r_0 $(1 \leq r_0 \leq k)$ denote the retrieval size used in building the tree. Given r_i, the top-r_i answers are identified at the ith round. When $\tau_U > k - r_0$, two or more rounds can be assigned, as in tournament sort. However, if $\tau_U \leq k - r_0$, the top-r_i answers can be identified at the ith round. Because τ_U is usually small, we mainly consider the second case, $\tau_U \leq k - r_0$.

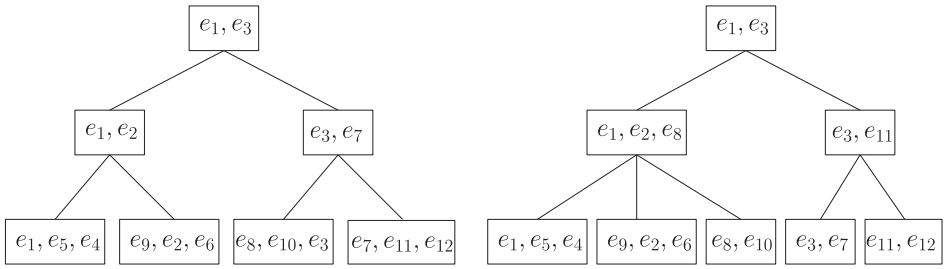


Fig. 4. Two tournaments with different buckets ($n = 12, \tau = 4,\ k = 2$).

Example 3 (Parameterized framework)**.** Continuing from Example 2, we are given $n = 12$, $\tau = 4$, and $k = 2$. Depending on $\mathcal{B}$ and $\mathcal{R}$, various execution plans can exist and their costs can vary. Fig. 3(a) depicts an execution plan with $\mathcal{B} = \langle\langle 4,3,3,2\rangle, \langle 3,1\rangle, \langle 2\rangle\rangle$. Given $\langle 4, 3, 3, 2\rangle$, $\mathcal{E}$ is divided into four buckets and the winners $\{e_1, e_2, e_3, e_{11}\}$ are used in the next round. Given $\mathcal{B}_2 = \langle 3, 1\rangle$, $\{e_1, e_2, e_3, e_{11}\}$ is divided into two buckets $\{e_1, e_2, e_3\}$ and $\{e_{11}\}$, and then $\{e_1, e_{11}\}$ is promoted to the next round. Finally, e_1 is identified as the top-1 answer. Fig. 3(b) depicts a tree that corresponds to the tournament in Fig. 3(a). For the tree to be built up, 17 questions are required over three rounds.

Next, the tree is incrementally updated with $\mathcal{R} = \langle 1, 2\rangle$. Given $r_1 = 2$, $\{e_4, e_2, e_{11}\}$ (that have lost to e_1) are identified as top-2 candidates. For $\{e_4, e_2, e_{11}\}$, three questions (e_4, e_2), (e_4, e_{11}), and (e_2, e_{11}) are asked. Finally, the top-2 answers are identified as $\{e_1, e_2\}$. In total, the execution plan generates 20 questions during four rounds. Similarly, given $\mathcal{R} = \langle 1, 3\rangle$, $\{e_4, e_2, e_{11}, e_9, e_6, e_3\}$ are considered top-3 candidates, and 15 questions are asked to the crowds. Compared to tournament sort, this requires more questions; however it reduces the number of rounds from seven to four.

To identify a cost-minimal execution plan, it is essential to select an *optimal* $\mathcal{B}$ and $\mathcal{R}$ that minimize the number of questions. The brute-force method is to enumerate all possible combinations for $\mathcal{B}$ and $\mathcal{R}$, and to select the optimal values. Because the search space is exponential, this is prohibitively expensive. In the next section, we explain how to efficiently select $\mathcal{B}$ and $\mathcal{R}$.

4. Crowd-enabled algorithm

In this section, we first present three methods to determine $\mathcal{B}$ and $\mathcal{R}$: *greedy, equi-sized*, and *dynamic programming*. We then propose four algorithms: GdyBucket, EquiBucket, EquiRange, and CrowdK, by selectively combining the three methods at each phase. When the answers of the crowd are imprecise, we also discuss a method to improve the accuracy of the top-k answers for a given execution plan.

4.1. Determining a sequence of bucket sizes

The number of questions in building a tree can differ from $\mathcal{B}$. Given $\mathcal{B}_i = (b_{i1}, \ldots, b_{i|\mathcal{B}_i|})$, the number of questions is calculated as $|\mathcal{Q}_i| = \sum_{j=1}^{|\mathcal{B}_i|} \binom{b_{ij}}{2}$. Given $\mathcal{B} = \langle \mathcal{B}_1, \ldots, \mathcal{B}_{\tau_B}\rangle$, the total number of questions is thus calculated as $\sum_{i=1}^{\tau_B} \sum_{j=1}^{|\mathcal{B}_i|} \binom{b_{ij}}{2}$.

Example 4 (Cost of building a tree)**.** We continue to illustrate Example 3. Given $n = 12$ and $\tau_B = 3$, Fig. 4 depicts two tournaments with different buckets. For each tournament, the number of questions is calculated as 15 and 14, respectively. Compared to the tournament in Fig. 3, we can further reduce the number of questions.

To minimize the number of questions, it is essential to select $\mathcal{B}$. The brute-force method is to exhaustively enumerate all possible values for $\mathcal{B}$. In that case, the time complexity increases exponentially with n. To address this problem, we develop two optimization methods: *horizontal partitioning* and *vertical partitioning*.

4.1.1. Determining horizontal bucket sizes

Suppose that an item set $\mathcal{E}$ is partitioned into m buckets, $\mathcal{B}_i = \langle b_{i1}, \ldots b_{im}\rangle$, at the ith round. We call this *horizontal partitioning*. Two conditions hold for b_{ij}: (i) $1 \le b_{ij} \le |\mathcal{E}|$ and (ii) $\sum_{j=1}^{m} b_{ij} = |\mathcal{E}|$. All possible cases of horizontal partitioning are $\binom{|\mathcal{E}'|-1}{m-1}$ which is overly costly to enumerate.

It is observed that the number of questions increases quadratically with b_{ij}. That is, the number of questions for $\mathcal{B}_i$ decreases as the sizes of the buckets are distributed as evenly as possible. Given $\mathcal{B}_i = \langle b_{i1}, \ldots b_{im}\rangle$, let $b_{i*}^{\min} = \min(b_{i1}, \ldots, b_{im})$ and $b_{i*}^{\max} = \max(b_{i1}, \ldots, b_{im})$ denote the minimum and the maximum value on $\mathcal{B}_i$, respectively. We formally state a key property for horizontal partitioning. (See proof in the Appendix.)

Theorem 1 (Horizontal optimization). *An item set $\mathcal{E}$ is partitioned into m buckets, $\mathcal{B}_i = \langle b_{i1}, \dots b_{im} \rangle$. If $b_{i*}^{min} = \lfloor |\mathcal{E}|/m \rfloor$ and $b_{i*}^{max} = \lceil |\mathcal{E}|/m \rceil$ hold, $\mathcal{B}_i$ minimizes $|\mathcal{Q}_i|$.*

Based on Theorem 1, we develop a horizontal partitioning method. Given $\mathcal{E}$, it is evenly distributed to m buckets and b_{i*}^{max} is computed by $\lceil |\mathcal{E}'|/m \rceil$. When $b_{i*}^{max} > |\mathcal{E}|$, some buckets can have surplus items. Let x denote the number of surplus items, *i.e.*, $x = mb_{i*}^{max} - n$. In this case, x buckets have b_{i*}^{min} items, and $|\mathcal{Q}_i|$ is computed by:

$$|\mathcal{Q}_i| = \binom{b_{i*}^{max}}{2} \times (m - x) + \binom{b_{i*}^{min}}{2} \times x, \tag{2}$$

where the time complexity is $O(m)$ because the number of buckets is m. (Algorithm 3 describes the horizontal partitioning.) Let $\mathcal{Q}(n, m)$ denote the number of questions using the proposed horizontal partitioning method. For example, when a set $\mathcal{E}$ of 12 items is evenly partitioned into 5 buckets, $\mathcal{B}_i = \langle 3, 3, 2, 2, 2 \rangle$, $\mathcal{Q}(12, 5)$ is 9. Note that $\mathcal{Q}(n, m)$ is also used for vertical partitioning.

4.1.2. Determining vertical bucket sizes

In addition to horizontal partitioning, $\mathcal{B}$ also depends on *vertical partitioning*. Because the winners at each bucket are only promoted to the next round, the number of items at the $(i+1)$th round is equal to the number of buckets partitioned at the ith round. If the proposed horizontal partitioning method is used at each round, the number of items at the $(i+1)$th round can be derived from the maximum bucket size at the ith round. That is, vertical partitioning can be used to compress a whole sequence of $\mathcal{B}$. Let $\widetilde{\mathcal{B}}$ denote a sequence of vertical partitioning, implying a sequence of maximum bucket sizes at each round, *i.e.*, $\widetilde{\mathcal{B}} = \langle \widetilde{\mathcal{B}}_1, \dots, \widetilde{\mathcal{B}}_{\tau_B} \rangle$ such that $\widetilde{\mathcal{B}}_i = b_{i*}^{\max}$. For example, given $\mathcal{B} = \langle \langle 4, 4, 4 \rangle, \langle 2, 1 \rangle, \langle 2 \rangle \rangle$, $\widetilde{\mathcal{B}}$ is represented by $\langle 4, 2, 2 \rangle$.

We now discuss how to determine $\widetilde{\mathcal{B}}$. Let $\mathcal{E}_i$ denote a set of items to be partitioned at the ith round, *e.g.*, $\mathcal{E}_1 = \mathcal{E}$. Basically, two conditions hold for $\widetilde{\mathcal{B}}_i$: (i) $2 \le \widetilde{\mathcal{B}}_i < |\mathcal{E}_i|$ and (ii) $n \le \prod_{i=1}^{\tau_B} \widetilde{\mathcal{B}}_i$. Initially, $|\mathcal{E}_1|$ is set by n. Because the winners at each bucket are only promoted to the next round, $|\mathcal{E}_{i+1}|$ $(1 \le i < \tau_B)$ is calculated as:

$$|\mathcal{E}_{i+1}| = \lceil \frac{|\mathcal{E}_i|}{\widetilde{\mathcal{B}}_i} \rceil. \tag{3}$$

Based on this equation, we first develop two heuristic methods to determine $\widetilde{\mathcal{B}}$: *greedy* and *equi-sized*. The greedy method is described as follows. For $2 \le \widetilde{\mathcal{B}}_i < |\mathcal{E}_i|$, the number of questions is calculated by $\mathcal{Q}(|\mathcal{E}_i|, \lceil |\mathcal{E}_i|/\widetilde{\mathcal{B}}_i \rceil)$ at the ith round. That is, $|\mathcal{E}_i|$ items are partitioned by $\lceil |\mathcal{E}_i|/\widetilde{\mathcal{B}}_i \rceil$ buckets. Because the greedy method focuses on minimizing the number of questions at the ith round, it is simple and fast. The time complexity is $O(\tau_B)$.

The equi-sized method assigns the same bucket sizes except for the last round. When $\widetilde{\mathcal{B}}_i = b$, $\widetilde{\mathcal{B}}$ is represented by $\langle b, \dots, b, \lceil n/b^{\tau_B - 1} \rceil \rangle$, where $b^{\tau_B - 1} < n$ holds. After generating all possible equi-sized buckets, the number of questions is calculated by $\sum_{i=1}^{\tau_B - 1} \mathcal{Q}(\lceil n/b^i \rceil, \lceil \lceil n/b^{i-1} \rceil / b \rceil) + \mathcal{Q}(\lceil n/b^{\tau_B - 1} \rceil, 1)$. Then, it selects b that minimizes the number of questions. The time complexity is thus $O(\tau_B n)$. Although the two methods can determine vertical buckets, they do not guarantee finding $\widetilde{\mathcal{B}}$ that minimizes the number of questions.

Alternatively, we propose a method using *dynamic programming*. Because the process of identifying $\widetilde{\mathcal{B}}$ can be divided to sub-problems, $\widetilde{\mathcal{B}}$ can be computed in a recursive manner. When a sequence of vertical partitioning is performed, it can be used to build another vertical partitioning. That is, it is possible to reuse pre-computed vertical partitioning.

Specifically, let B and V denote two matrices for storing a sequence of vertical bucket sizes and the number of questions, respectively. Given B, $B[i, j]$ represents an entry that stores a sequence of bucket sizes with respect to i rounds and j items. Given $B[i, j]$, the current bucket sizes are derived from the previous round. If $B[i, j]$ is feasible, $V[i, j]$ stores the total number of questions for $B[i, j]$. Given $B[i, j]$, $V[i, j]$ is computed by the following recursive equation. (In Example 5, we will explain a detailed example of how to compute B and V.)

- *Initial step* $(i = 1)$: $|\mathcal{E}_i|$ and $\widetilde{\mathcal{B}}_i$ are set by n and j such that $2 \le j \le n$, and $V[i, j]$ is computed by $\mathcal{Q}(n, \lceil \frac{n}{j} \rceil)$.
- *Recursive step* $(2 \le i \le \tau_B)$: $|\mathcal{E}_i|$ and $\widetilde{\mathcal{B}}_i$ are derived from the previous round. At the ith round, the feasible range of $|\mathcal{E}_i|$ is $[2^{i-1} + 1, \frac{n}{2^{\tau_B - i}}]$ under the condition for $\widetilde{\mathcal{B}}$. Given $|\mathcal{E}_i| = j$, the range of bucket sizes is $[2, j - 1]$. Therefore, $V[i, j]$ is computed by $\min_{b \in [2, j-1]} V[i - 1, \lceil \frac{j}{b} \rceil] + \mathcal{Q}(j, \lceil \frac{j}{b} \rceil)$. For $B[i, j]$, the current bucket size is appended to the previous sequence of vertical bucket sizes. Therefore, $B[i, j]$ is set by concatenating b to $B[i - 1, \lceil \frac{j}{b} \rceil]$.

In the recursive step, we prove that the *principle of optimality* holds. Using this property, we can choose $\widetilde{\mathcal{B}}$ that minimizes the number of questions. Formally, we state vertical optimization that minimizes the number of questions.

Lemma 1. *For $2 \le i \le \tau_B$ and $2^{i-1} + 1 \le j \le n$, $V[i, j] = \min_{b \in [2, j-1]} V[i - 1, \lceil \frac{j}{b} \rceil] + \mathcal{Q}(j, \lceil \frac{j}{b} \rceil)$.*

Theorem 2 (Vertical optimization). *Given n and τ_B, $\widetilde{\mathcal{B}}$ minimizes $|\mathcal{Q}_B|$, if $\widetilde{\mathcal{B}}$ is determined by $B[\tau_B, n]$.*

We describe the pseudo-code of the proposed dynamic programming (Algorithm 1). For efficiency, it only generates feasible values for $|\mathcal{E}_i|$ at each round. For the ith round $(1 \le i < \tau_B)$, the range of $|\mathcal{E}_i|$ is $[2^{i-1} + 1, \frac{n}{2^{\tau_B - i}}]$ (lines 4–7). As

Algorithm 1: VerticalPartitioning(n, τ_B, r_0).

Input: the number of items n, the number of rounds τ_B, initial range r_0
Output: a sequence of bucket sizes $\langle \widetilde{\mathcal{B}}_1, \ldots, \widetilde{\mathcal{B}}_{\tau_B} \rangle$

1 $\tau_B \leftarrow \min(\tau_B, \lceil \log_2 n \rceil)$;
2 Let V and B be $\tau_B \times n$ matrices;
3 **for** $i \leftarrow 1$ ***to*** τ_B **do**
4 $min\_b \leftarrow 2^{i-1} + 1$, $max\_b = \lceil \frac{n}{2^{\tau_B - i}} \rceil$
5 **if** $i = \tau_B$ **then**
6 $min\_b \leftarrow n$ /* the degenerate case when $i = \tau_B$ */
   /* minimum and maximum values of $|\mathcal{E}_i|$ */;
7 **for** $j \leftarrow min\_b$ ***to*** max_b **do**
8 **if** $i = 1$ **then**
9 $V[i, j] \leftarrow \binom{j}{2}$, $B[i, j] \leftarrow j$
10 **else**
11 $V[i, j] \leftarrow \infty$
12 **for** $b \leftarrow 2$ ***to*** $j - 1$ **do**
13 $l \leftarrow \lceil \frac{j}{b} \rceil$ /* when $|\mathcal{E}_i| = j$ and $\widetilde{\mathcal{B}}_i = b$ */;
14 **if** $V[i-1, l]$ *is null* **then**
15 Break
16 **if** $i = \tau_B$ and $r_0 > 1$ **then**
   /* the last round when $i = \tau_B$ */
17 Build a tree $\mathcal{T}$ for $B[i-1, l]$
18 $|\mathcal{Q}_i| \leftarrow$ EstimateUpdateCost($\mathcal{T}$, r_0)
19 **else**
20 $|\mathcal{Q}_i| \leftarrow \mathcal{Q}(j, l)$;
21 **if** $V[i-1, l] + |\mathcal{Q}_i| < V[i, j]$ **then**
22 $V[i, j] \leftarrow V[i-1, l] + |\mathcal{Q}_i|$
23 $B[i, j] \leftarrow$ concatenate b to $B[i-1, l]$
24 **return** $B[\tau_B, n]$

the initial step, $V[1, j]$ and $B[1, j]$ are computed by $\binom{j}{2}$ and j, respectively (line 9). When $2 \le i \le \tau_B$, $|\mathcal{E}_i|$ and $\widetilde{\mathcal{B}}_i$ are set by j and b (lines 7 and 12). Given $j = |\mathcal{E}_i|$ and $b = \widetilde{\mathcal{B}}_i$, $l = |\mathcal{B}_i|$ is computed by $\lceil \frac{j}{b} \rceil$ (line 13). After $\mathcal{Q}(j, l)$ is computed, $V[i, j]$ and $B[i, j]$ are recursively computed (lines 21–23). (As an exceptional case, *i.e.*, $r_0 > 1$, we explain how to compute $|\mathcal{Q}_i|$ in Section 4.2.) Finally, it returns $\widetilde{B} = B[\tau_B, n]$. Because the size of the range in $|\mathcal{E}_i|$ is at most $\lceil \frac{n}{2^{\tau_B - i}} \rceil$ items at the ith round, and only $\lceil \frac{n}{2^{\tau_B - i + 1}} \rceil$ items are generated at $(i-1)$th round, the time complexity is bounded to $O(n^2)$.

Example 5 (Dynamic programming)**.** Continuing from Example 4, we are given $n = 12$, $\tau_B = 3$, and $r_0 = 1$. Fig. 5 describes two matrices V and B at each round. When $i = 1$ and $2 \le j \le 3$, $V[1, j]$ and $B[1, j]$ are initialized as $\binom{j}{2}$ and j. At the next round, $V[2, j]$ and $B[2, j]$ ($3 \le j \le 6$) are computed. When calculating $V[2, 6]$, it generates $V[1, 2] + \mathcal{Q}(6, 2) = 7$ and $V[1, 3] + \mathcal{Q}(6, 3) = 6$ by reusing the pre-identified sequences. Because the latter is smaller than the former, $V[2, 6]$ and $B[2, 6]$ are set as 6 and $\langle 2, 3 \rangle$, respectively. Similarly, when computing $V[3, 12]$, the feasible range of b is [2, 5]. We can reuse $V[2, 3]$, $V[2, 4]$, and $V[2, 5]$ in determining b. When $V[2, 3]$ is chosen, $V[3, 12]$ is minimized. Fig. 6 depicts the corresponding tree for $\widetilde{\mathcal{B}} = \langle 2, 2, 3 \rangle$. Compared to the other tournaments in Fig. 4, it only generates 12 questions and is comparable to the tournament sort that builds a tournament with 11 questions in Fig. 1.

4.2. Determining a sequence $\mathcal{R}$ of retrieval sizes

After a tree is built up, we should update the tree incrementally to identify the top-k answers. Given τ_U rounds, the tree is updated by a sequence $\mathcal{R} = \langle r_0, \ldots, r_{\tau_U} \rangle$ of retrieval sizes, called a *range*. In this process, the cost of updating the tree highly depends on $\mathcal{R}$. It is thus essential to choose $\mathcal{R}$ that minimizes the number of questions. Therefore, we address the following two issues:

1. *Removing unnecessary questions:* It is important to only generate necessary questions for identifying the top-r_i answers. We thus fully exploit the pre-obtained precedences between items.

$V[i,j]$	2	3	4	5	6	7	...	11	12
1	1	3	-	-	-	-	...	-	-
2	-	2	3	5	6	-	...	-	-
3	-	-	-	-	-	-	...	-	12

(a) 3×12 matrix V for $|\mathcal{Q}_B|$

$B[i,j]$	2	3	4	5	6	7	...	11	12
1	$\langle 2 \rangle$	$\langle 3 \rangle$	-	-	-	-	...	-	-
2	-	$\langle 2,2 \rangle$	$\langle 2,2 \rangle$	$\langle 2,3 \rangle$	$\langle 2,3 \rangle$	-	...	-	-
3	-	-	-	-	-	-	...	-	$\langle 2,2,3 \rangle$

(b) 3×12 matrix B for bucket sizes

Fig. 5. Illustration of finding vertical bucket sizes ($n = 12$, $\tau = 3$, $r_0 = 1$).

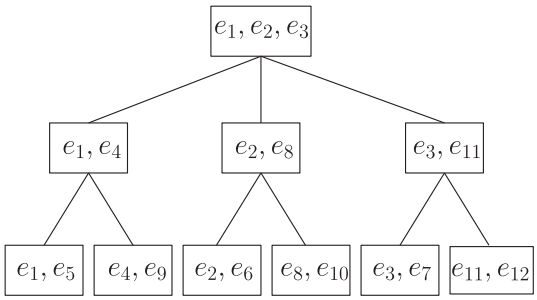


Fig. 6. Tournament result for Fig. 5.

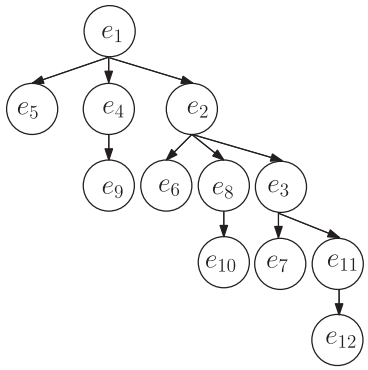
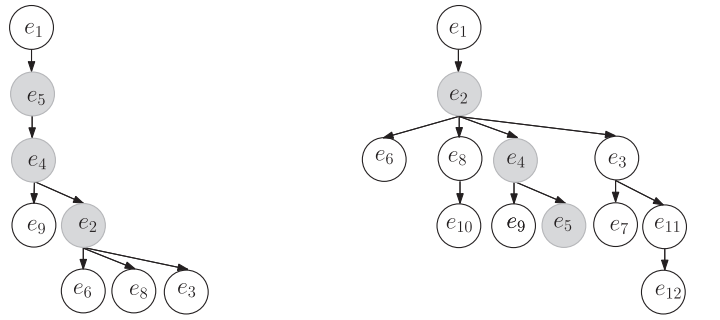


Fig. 7. Tree that corresponds to tournament in Fig. 6.

2. *Predicting the answers of questions:* Because the answers of questions are unpredictable, it is difficult to obtain the updated tree deterministically. Therefore, it is non-trivial to calculate the number of questions in updating the tree.

To address these issues, we use a tree $\mathcal{T}$ to identify the precedences between items. If $e_i \succ e_j$, a directed edge in $\mathcal{T}$ is connected from e_i to e_j, implying a *parent-child relationship*. For brevity, some edges can be skipped if the precedences between items can be inferred by transitivity. Let *Ancestor*(e_i) denote a set of *ancestors* of e_i. The number of ancestors |*Ancestor*(e_i)| is thus equal to the number of more preferred items, which corresponds to the level of the node in $\mathcal{T}$. Therefore, if |*Ancestor*(e_i)| is smaller than retrieval size r_i, e_i can be in the top-r_i candidates.

Example 6 (Updating the tournament tree)**.** Fig. 7 depicts a tree that corresponds to the tournament in Fig. 6. The root node e_1 represents the top-1 item. Based on the tree, we can identify top-r_i candidates according to the level of the node. For example, when $r_1 = 2$, $\{e_2, e_4, e_5\}$ is a set of top-2 candidates. Similarly, when $r_1 = 3$, $\{e_3, e_6, e_8, e_9\}$ is appended to the top-3 candidates.



(a) best case ($e_5 \succeq e_4 \succeq e_2$) (b) worst case ($e_2 \succeq e_4 \succeq e_5$)

Fig. 8. Two possible trees updated for directed tree in Fig. 7.

Next, we explain how to generate necessary questions for the top-r_i candidates. The simplest method is to enumerate all pair-wise questions for top-r_i candidates. Given $r_1 = 3$ in Fig. 7, the top-3 candidates are $\{e_2, e_3, e_4, e_5, e_6, e_8, e_9\}$, and $\binom{7}{2} = 21$ questions are required. However, it is found that unnecessary questions can be generated in two cases: (i) (e_4, e_9), (e_2, e_6), (e_2, e_8), and (e_2, e_3) are redundant as their precedences can be obtained from the previous rounds, and (ii) (e_6, e_9), (e_8, e_9), and (e_3, e_9) are unnecessary, because they are irrelevant for determining the top-3 answers. Given (e_6, e_9), the ancestor sets of e_6 and e_9 can be derived as $\{e_1, e_2\}$ and $\{e_1, e_4\}$, respectively. This means that only one of e_6 and e_9 can be in the top-3 answers as either $\{e_1, e_2\}$ or $\{e_1, e_4\}$ has been identified as top-2 answers. That is, (e_6, e_9), (e_8, e_9), and (e_3, e_9) are irrelevant for determining the top-3 answers. By removing seven questions in the two cases, it is sufficient to ask 14 questions to evaluate the top-3 answers, *i.e.*, $\binom{7}{2} - 7$.

Lemma 2. *When r_i and $\mathcal{T}$ are given, (e_i, e_j) can be skipped, if (1) $e_i \in$ Ancestor(e_j), (2) $e_j \in$ Ancestor(e_i), or (3) $|$Ancestor(e_i) $\cup$ Ancestor(e_j)$| \geq r_i$.*

The next challenging issue is how to estimate the number of questions when updating a tree. Given $k = 3$ and $\tau_U = 2$, suppose that $\mathcal{R}$ is set as $\langle 1, 2, 3\rangle$ in Fig. 7. Given $r_1 = 2$, three questions, (e_2, e_4), (e_2, e_5), and (e_4, e_5) are required. Because the top-2 candidates are deterministic, it is easy to calculate the number of questions. However, the form of the updated tree for $r_2 = 3$ is determined by the answers of the previous questions (e_2, e_4), (e_2, e_5), and (e_4, e_5). Therefore, it is impossible to build a deterministic tree without knowing the precedences between items.

One possible solution is to estimate the average number of questions by considering all updated trees. Given $\{e_2, e_4, e_5\}$, the number of possible answers is six, *i.e.*, 3!. Depending on the answer, different trees can be updated. Fig. 8 depicts two possible updated trees for the top-2 answers. In the best case, no questions are required for the top-3 answers because e_4 is the only top-3 candidate. However, in the worst case, $\{e_3, e_4, e_6, e_8\}$ can be the top-3 candidates, and six questions are required. For each updated tree, the number of questions can be calculated as zero, zero, one, one, six, and six. The average number of questions is thus $\frac{14}{6} = 2.3$.

Although it is feasible for estimating the number of questions, generating all possible updated trees is overly exhaustive. This problem is reduced by counting the number of different ways to perform a *topological sort* for a set of nodes. This is known as a #P-complete problem [7]. When the number of top-r_i candidates is large, it becomes impractical.

Alternatively, we use an approximation algorithm using *random sampling*. Let π_{rand} denote a random latent scoring function for n items. When π_{rand} is given, we can update the tree in a deterministic manner. Based on a randomized scheme, we generate δ random functions and compute the average number of questions for updating a tree. It can be viewed as δ iterations.

Existing work [8] demonstrates that the randomized algorithm can be derived as a *fully polynomial randomized approximation scheme*. That is, as δ is greater, it is closer to the average number of questions; however, this incurs more computation. In our experiments, we empirically set δ to five. Although it only covers a small part of all possible cases, from our empirical study, the estimated number of questions is approximately close to the actual number of questions in performing a query execution plan.

Lastly, we explain how to determine $\mathcal{R}$. Similar to building a tree, three methods can be used: *greedy, equi-sized*, and *dynamic programming*. The key difference is that $\mathcal{R}$ should consider k and τ_U instead of n and τ_B. Unlike the tree-building step, because the tree updating procedure is not deterministic, dynamic programming does not preserve the principle of optimality. However, it is still useful to determine $\mathcal{R}$ that minimizes the number of questions.

Algorithm 2: BuildTournament(n, τ_B, r_0).

Input: the number of items n, the number of rounds τ_B, initial range r_0
Output: a sequence of bucket sizes $\mathcal{B}$

```
1 Let B be a sequence of bucket sizes;
2 B̃ ← VerticalPartitioning(n, τ, r_0);
3 |E_i| ← n;
4 for i ← 1 to τ_B do
5 |   B_i ← HorizontalPartitioning(|E_i|, ⌈|E_i|/B̃_i⌉);
6 |   |E_i| ← ⌈|E_i|/B̃_i⌉;
7 return B
```

4.3. Algorithm description

In this section, we propose a crowd-enabled top-k algorithm, called CrowdK. This algorithm consists of two steps: (i) building a tree and (ii) updating the tree. For each step, CrowdK adopts dynamic programming. Note that, when $\tau \leq 2$ or $k = 1$, the second step can be skipped.

Algorithm 2 describes the pseudocode of identifying $\mathcal{B}$ in building a tree. We first identify $\widetilde{\mathcal{B}}$ using Algorithm 1 (line 2) and then perform horizontal partitioning for $\widetilde{\mathcal{B}}_i$ at each round (lines 3–6). Once $\mathcal{B}$ is identified, the tree can be built for π_{rand} and the number of questions can be calculated. Recall that when $\mathcal{R}$ is only represented by $\langle r_0 \rangle$, *e.g.*, max queries, the process of updating the tree can be skipped. Therefore, it is sufficient to identify $\mathcal{B}$ that minimizes the number of questions in building a tree.

Algorithm 3 describes the pseudo-code of horizontal optimization. Given the number of items n and the number of buckets m, $|\mathcal{E}_i|$ is divided by m to compute an even bucket size, *i.e.*, $b = \lceil |\mathcal{E}_i|/m \rceil$ (line 2). When $r = m \times b - n$ is greater than zero, $b - 1$ items are distributed for r (lines 5–6). Otherwise, b items are evenly distributed (lines 7–8). For m buckets, the items are distributed to minimize the difference between the maximum and the minimum bucket sizes. The time complexity of horizontal optimization is $O(m)$.

Algorithm 4 presents the overall procedure of identifying $\mathcal{B}$ and $\mathcal{R}$. Similar to $\mathcal{B}$, enumerating all possible cases for $\mathcal{R}$ is overly expensive, *i.e.*, $\binom{k-1}{\tau_U}$. We thus exploit dynamic programming for $\mathcal{R}$, where two $\tau \times k$ matrices V and T are used. Let $V[i, j]$ denote the number of questions such that $\tau = i$ and $k = j$. Similarly, $T[i, j]$ indicates a tree such that $\tau = i$ and $k = j$. This first performs Algorithm 2 to identify $\mathcal{B}$ (line 6). If $\tau \leq 2$ or $k = 1$, identifying $\mathcal{R}$ can be skipped (line 9). Let $C(\mathcal{T})$ denote the number of estimated questions used in building $\mathcal{T}$. When $r_i = j$, the tree $T[i-1, l]$ at the previous round ($1 \leq l \leq j-1$) is used for determining $\mathcal{R}$ (line 12). In this process, it chooses a tree that minimizes the number of questions (lines 10–15). Finally, $\mathcal{B}$ and $\mathcal{R}$ can be identified by backtracking $T[\tau, k]$ (lines 16). Because EstimateUpdateCost() can incur $O(n^2)$ in the worst case, the time complexity of the tree-updating step in Algorithm 4 is $O(\tau k^2 n^2)$.

Algorithm 5 describes the pseudo-code for estimating the number of questions in updating a tree. As input, it takes a tournament tree $\mathcal{T}$ and a range r_i. We first identify a set of the top-r_i candidates using the level of the tree (line 2). For each pair-wise question, we check if pair (e_i, e_j) satisfies the three conditions: (i) $e_i \notin Ancestor(e_j)$, (ii) $e_j \notin Ancestor(e_i)$, and (iii) $|Ancestor(e_i) \cup Ancestor(e_j)| < u_i$ (lines 4–6). If so, (e_i, e_j) is added to $\mathcal{Q}_i$ (line 7). Finally, this returns the number of question $|\mathcal{Q}_i|$ at the ith round (line 8). The time complexity is $O(n^2)$, where the number of candidate items $|C|$ reaches n. However, $|C|$ is usually smaller than n.

Algorithm 3: HorizontalOptimization(n, m).

Input: the number of items n, the number of buckets m
Output: a sequence of horizontal bucket sizes $\langle b_{i1}, \ldots, b_{im} \rangle$

```
1 Let B_i be a sequence of size m
2 b ← ⌈n/m⌉          /* even bucket size                    */;
3 r ← mb − n         /* # of remained items                 */;
4 for j ← 1 to m do
5 |   if j ≤ r then
6 |   |   b_ij ← b − 1   /* minimum bucket size              */;
7 |   else
8 |   |   b_ij ← b       /* maximum bucket size              */;
9 return B_i
```

Algorithm 4: CrowdK(n, τ, k).

Input: the number of items n, the number of rounds τ, retrieval size k
Output: a cost-optimal execution plan A

Let V and T be $\tau \times k$ matrices
Let π_{rand} be a list of n items with random function scores
Let $C(\mathcal{T})$ be the number of questions used in $\mathcal{T}$
for $i \leftarrow 2$ ***to*** τ **do**
  for $j \leftarrow 1$ ***to*** k **do**
    /* Build a tree for $\tau_B = i$ and $r_0 = j$ */
    $\mathcal{B} \leftarrow$ BuildTournament(n, i, j)
    $\mathcal{T} \leftarrow$ build a tree for $\mathcal{B}$ and π_{rand}
    $V[i, j] \leftarrow C(\mathcal{T})$, $T[i, j] \leftarrow \mathcal{T}$
    if $i > 2$ *and* $j > 1$ **then**
      $V[i, j] \leftarrow \infty$
      for $l \leftarrow 1$ ***to*** $j - 1$ **do**
        /* Estimate $|\mathcal{Q}_i|$ with $T[i-1, l]$ and j */
        $|\mathcal{Q}_i| \leftarrow$ EstimateUpdateCost($T[i-1, l]$, j)
        if $C(T[i-1, l]) + |\mathcal{Q}_i| \leq V[i, j]$ **then**
          $V[i, j] \leftarrow C(T[i-1, l]) + |\mathcal{Q}_i|$
          $T[i, j] \leftarrow$ update the tree for $T[i-1, l]$ and π_{rand}
$A \leftarrow$ find $\mathcal{B}$ and $\mathcal{R}$ by backtracking $T[\tau, k]$
return A

Algorithm 5: EstimateUpdateCost($\mathcal{T}$, r_i).

Input: a tournament tree $\mathcal{T}$, a range r_i

Let $\mathcal{Q}_i$ be a set of questions at ith round
$C \leftarrow$ a set of candidates for the top-r_i answers
foreach $(e_i, e_j) \in C \times C$, $i \neq j$ **do**
  if $e_i \in Ancestor(e_j)$ *or* $e_j \in Ancestor(e_i)$ **then**
    Continue;
  else if $|Ancestor(e_i) \cup Ancestor(e_j)| < u_i$ **then**
    Append (e_i, e_j) into $\mathcal{Q}_i$
return $|\mathcal{Q}_i|$

Table 3
CrowdK and its representative variants (DP indicates dynamic programming).

	GdyBucket	EquiBucket	EquiRange	CrowdK
Building method	Greedy	Equi-sized	DP	DP
Updating method	DP	DP	Equi-sized	DP
Time complexity for building	$O(\tau)$	$O(\tau n)$	$O(n^2)$	$O(n^2)$
Time complexity for updating	$O(\tau k^2 n^2)$	$O(\tau k^2 n^2)$	$O(\tau k n^2)$	$O(\tau k^2 n^2)$

While Algorithm CrowdK exploits dynamic programming in both building and updating a tree, we can also use other methods, greedy and equi-sized. Table 3 describes several notable variants and the corresponding complexity. Although these variants incur more monetary costs than CrowdK, they tend to run faster. (In Section 5, we evaluate the trade-off between our proposals.)

4.4. Improving the quality of answers

When the answers of the crowd are imprecise, we control the number ω of workers per question to enhance the accuracy of the top-k answers. Assuming that the accuracy of the questions is independent, improving the accuracy of the questions is orthogonal to our original problem formulation. We can thus apply various algorithms to tune the number of workers per question by considering the importance of the questions and the proficiency of the workers as discussed in [17,26,32]. (In a

real crowdsourcing system such as Amazon Mechanical Turk, because it does not provide various options of selecting crowd workers, it is only possible to consider the importance of the questions.)

As the simplest solution, we first suppose that workers correctly respond with an equal probability p without considering the importance of the questions. This assumption is also known as a *constant error model* [15]. Given a question (s, t), the probability that i $(0 \leq i \leq \omega)$ correct answers are returned is represented by a binomial distribution, and the probability of obtaining correct answers is calculated by *majority voting*. Given p, the accuracy θ of the pair-wise question is computed by:

$$\theta = \sum_{i=\lceil \frac{\omega}{2} \rceil}^{\omega} \binom{\omega}{i} p^i (1-p)^{\omega - i}. \tag{4}$$

Given θ, the precision of top-k answers can be estimated by a probabilistic model as discussed in [38] and ω can be tuned empirically. Although this method is effective for improving the accuracy of the questions, it neglects to consider other factors such as the importance of the questions and the difficulty of the questions.

Alternatively, we develop a heuristic assignment method to reflect the importance of the questions for a given tournament. Specifically, because the questions at the higher levels of a tree can be more important to determine top-k answers, the "utility" of questions can be designed by a *variable error model* [12]. That is, we can assign the number of workers depending on the level of the tree. This means that, as the level is close to the root, the importance of the questions can be high. Based on this observation, we allocate the number of workers per question in a dynamic manner. For instance, we categorize all questions into three classes. Let $|\mathcal{Q}|$ be the total number of questions and $l(s, t)$ be the position of a question (s, t) in a total question set $\mathcal{Q}$. Given a question (s, t), we first allocate a less number of workers for questions located at the lower levels, *i.e.*, 33% of the questions close to the leaf nodes. We then allocate a greater number of workers for questions located at the higher levels, *i.e.*, 33% of the questions close to the root. That is, when the default value for ω is five, we can dynamically assign ω as follows:

$$\omega = \begin{cases} 3, & if \;\; l(s,t) < |\mathcal{Q}| \cdot 1/3 \\ 5, & if \;\; |\mathcal{Q}| \cdot 1/3 \leq l(s,t) < |\mathcal{Q}| \cdot 2/3 \\ 7, & if \;\; |\mathcal{Q}| \cdot 2/3 \leq l(s,t) \end{cases}$$

This partitioning can be easily extended for three and more classes.[3] In Section 5.1, we compare the accuracy of the proposed static and dynamic methods, denoted as CrowdK (Static) and CrowdK (Dynamic).

Example 7 (Dynamic voting). Fig. 6 depicts the tournament result of Algorithm CrowdK. When we assign five workers per question with static voting, the total number of questions is 60 $(= 5 \times 12)$. However, we can assign a different number of workers per question depending on the level of the tournament. We first assign three workers per question at the lowest level. Then, five and seven workers are assigned per question at the middle level and high level, respectively. In this case, the total number of questions is 54 $(= 3 \times 6 + 5 \times 3 + 7 \times 3)$. Compared to the previous static voting, the total number of questions is similar. However, because the dynamic voting distinguishes the importance of the questions, the accuracy of the top-k queries can be improved.

5. Experiments

In this section, we conduct extensive experiments to validate the monetary cost, the accuracy of top-k answers, and running time of the proposed algorithms. We first compare them with synthetic datasets with various parameter settings (Section 5.1). We then evaluate the proposed algorithm with Amazon Mechanical Turk using real-life datasets (Section 5.2).

To evaluate the monetary cost, we measure the *number of questions* of the execution plan generated by the proposed algorithms. To quantify the accuracy of the top-k answers, three metrics are used and cross-checked: *precision at k, mean reciprocal rank (MRR)*, and *normalized discounted cumulative gain (NDCG)*.[4] As the scores of these metrics are closer to 1.0, the top-k answers become more accurate. For efficiency, we also measure the *running time* of the algorithms.

We compare the proposed algorithms, GdyBucket, EquiBucket, EquiRange, and CrowdK with Exhaustive. The Exhaustive algorithm enumerates all possible combinations of vertical bucket sizes and ranges over the parameterized framework, and chooses the values of buckets and ranges that minimize the number of questions. Given a sequence of vertical bucket sizes, this also uses the proposed horizontal partitioning. When the precedence of items is given, Exhaustive can minimize the number of questions. However, it is excessively much slower than our proposals because the number of possible cases increases exponentially with n and k, *i.e.*, $\binom{n}{\tau_B}$ and $\binom{k-1}{\tau_U}$ (Fig. 15). For all algorithms, when the answers of workers were erroneous, an *iterative strategy* [21] was used to resolve the conflict of answers. Being orthogonal to the proposed algorithms, other aggregation methods could be used.

All experiments were conducted in Windows 7 running on an Intel Core i7 950 3.07 GHz CPU with 16 GB RAM. All algorithms were implemented in C++. We performed ten executions for each parameter setting and reported the average values.

[3] Although it is interesting to optimize both the accuracy of top-k answers and monetary cost using a more sophisticated accuracy model, it can incur a more complicated problem formulation. In this paper, we thus leave the extension as our future work.

[4] As all three metrics showed similar patterns, in the interest of space, we only report NDCG in this paper.

Table 4
Parameter settings for synthetic datasets.

Parameters	Range	Default
Cardinality n	2K, 4K, 6K, 8K, 10K	4K
# of rounds τ	4, 5, 6, 7, 8	6
Retrieval size k	2, 4, ..., 20	10
# of workers per question ω	3, 5, 7, 9	5
Prob. of correct answers p	0.7, 0.8, 0.9	0.8

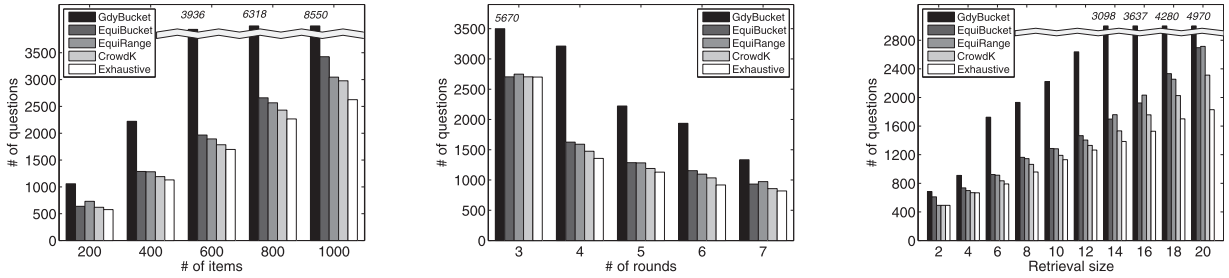


Fig. 9. Comparison of varying the number of questions in small-scale datasets.

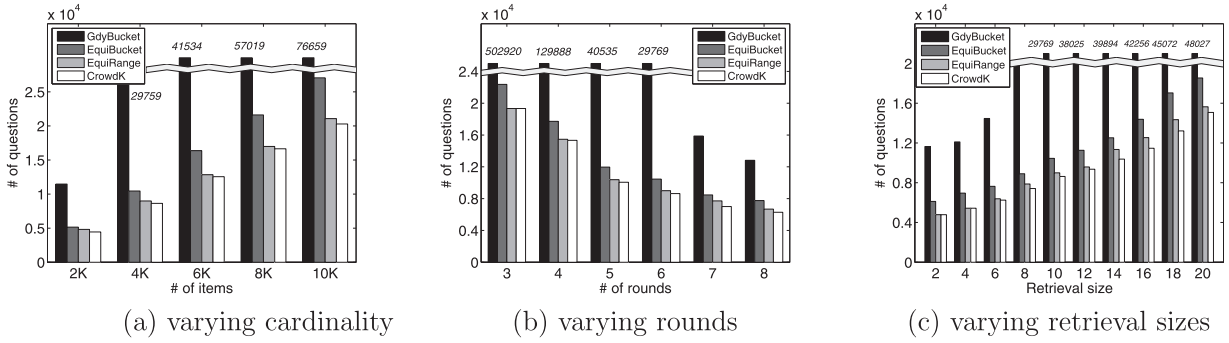


Fig. 10. Comparison of varying the number of questions in large-scale datasets.

5.1. Evaluation with synthetic datasets

We first evaluate the proposed algorithms over synthetic datasets with *five* parameters: the number n of items, the number τ of rounds, retrieval size k, the number ω of workers, and the probability p of obtaining correct answers. Given n items, the ground truth for the latent function scores was randomly generated. We assumed that large scores for items were preferred. Table 4 summarizes the parameter values used in synthetic datasets.

Monetary cost: When the cost per question and the number of workers are fixed, the total cost is proportional to the number of questions asked to the crowd. When the latent function scores are known, Exhaustive minimizes the number of questions. Because Exhaustive is overly slow for large-scale synthetic datasets, we compared all five algorithms including Exhaustive in small-scale datasets (*i.e.*, $n = [200, 1000]$, $\tau = [3, 6]$, $k = [2, 20]$). Then, we evaluated our four proposals in large-scale datasets. In the crowdsourcing setting, because the number of items is closely related to the monetary cost, the number of items is controlled using pre-processing, *e.g.*, using Boolean condition in Example 1. Therefore, a set of 10,000 items was sufficiently large to evaluate the crowd-enabled algorithms.

Fig. 9 depicts the number of questions in the small-scale datasets. The default parameter values were $n = 400$, $\tau = 5$, and $k = 10$. (CrowdK is CrowdK (Static) that employs conventional assignment method with the constant error model in Fig. 14.) Note that CrowdK is always the best algorithm among the four proposals, being closest to the minimum bound achieved by Exhaustive in all parameter settings. As k increases, the difference between Exhaustive and the others increases, yet CrowdK remains close to Exhaustive. As τ increases, all algorithms can avoid asking unnecessary questions by leveraging the pre-obtained precedences between items (Fig. 9b). Inversely, as n or k increases, the number of questions increases (Fig. 9a and c). Overall, GdyBucket is the worst algorithm, generating 2–3 times more questions than CrowdK does. This is because the building process of GdyBucket deteriorates as n or k increases. EquiBucket and EquiRange have a comparable performance in the small-scale datasets.

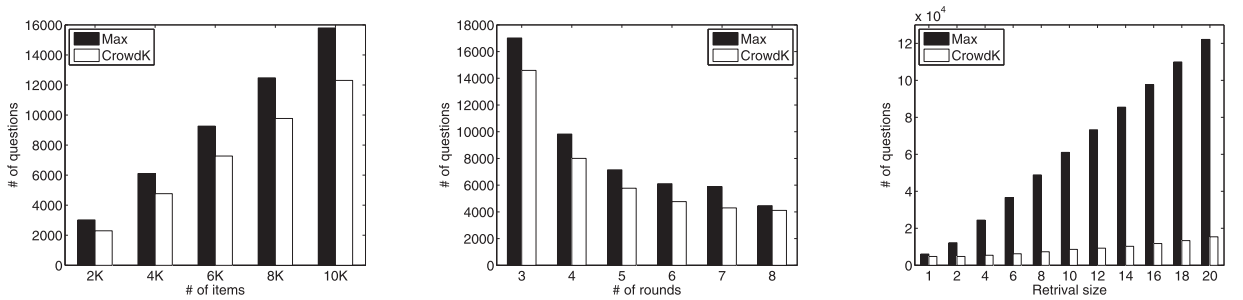


Fig. 11. Comparison of varying the number of questions between Max and CrowdK.

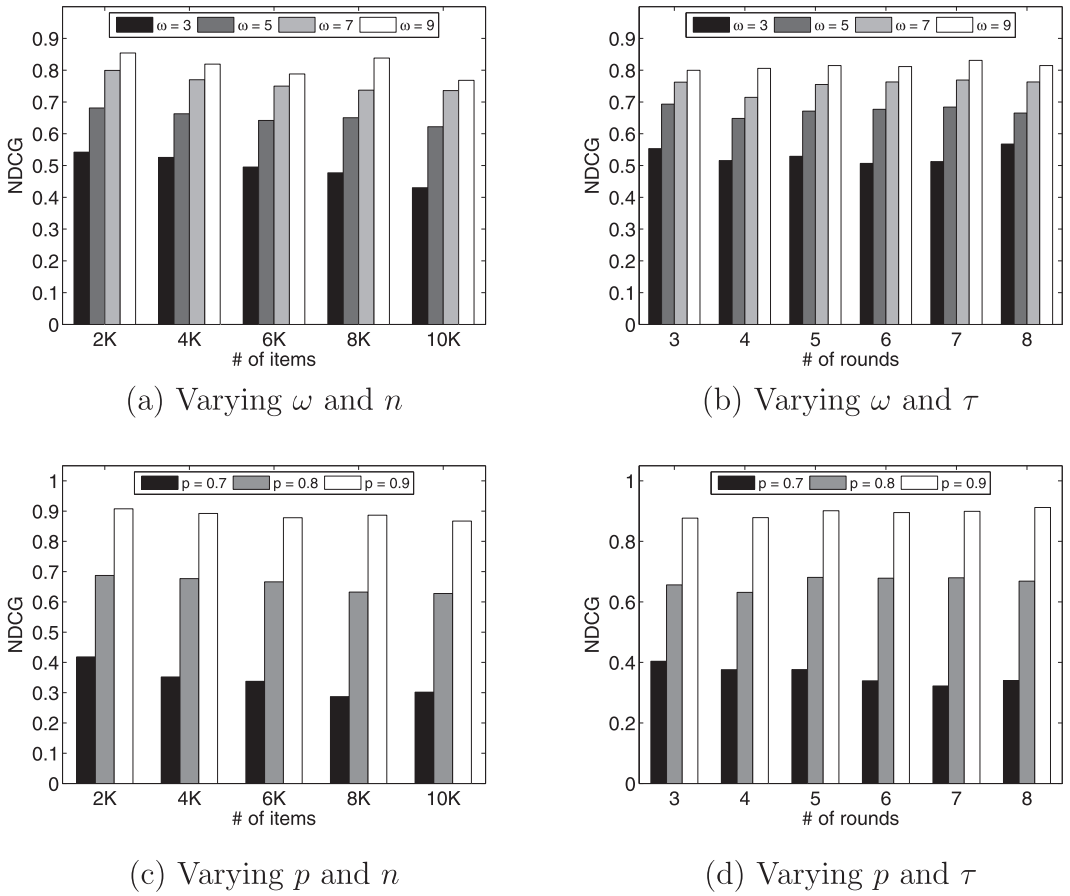


Fig. 12. Accuracy of varying number ω of workers and probability p in CrowdK.

Fig. 10 depicts the number of questions in the large-scale datasets. Compared to the other algorithms, CrowdK also yields the minimum questions . Unlike the small-scale datasets, EquiRange shows superior performance compared to EquiBucket as n or k increases (Fig. 10a and c). This means that equi-sized ranges are more effective than equi-sized buckets as an approximate method of CrowdK. GdyBucket still demonstrates the worst performance as in the small-scale datasets.

Lastly, we compare CrowdK with Max proposed in [38] which is one of the closest existing works (Fig. 11). (Note that, although [39] addresses the max queries, the only difference between [38] and [39] is to employ the estimate function for latency. In our problem formulation, because the latency is measured by the number of rounds, [39] is similar to [38].) As a more favorable setting for Max, k was set as one by default in Fig. 11(a) and (b). Because [38] addresses the different problem of optimizing the quality of the answers under the constraint for monetary cost and latency, it was impossible to directly compare this with CrowdK. We thus modified Max to address our problem by removing the constraint for

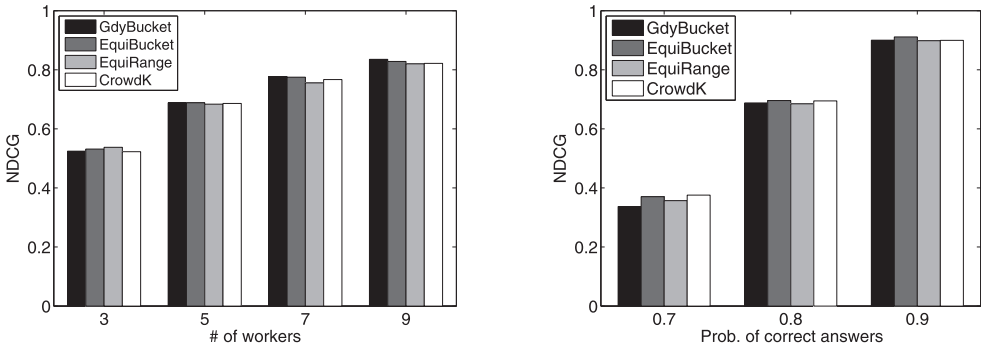


Fig. 13. Accuracy comparison of algorithms over varying number ω of workers and probability p in CrowdK.

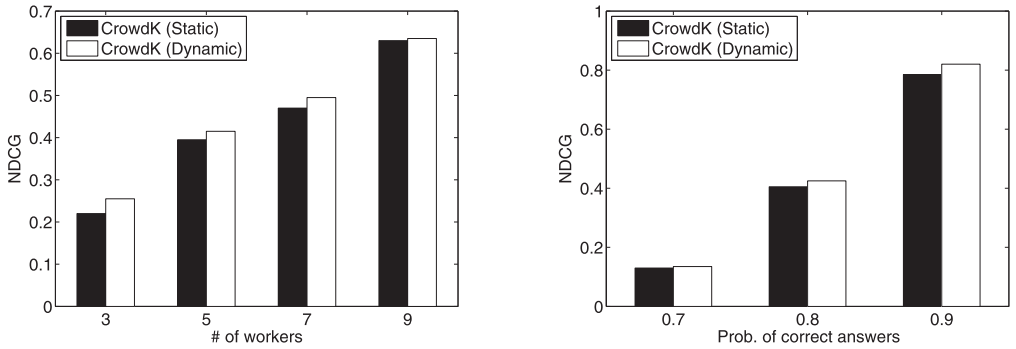


Fig. 14. Accuracy comparison of CrowdK (Static) and CrowdK (Dynamic) over varying number ω of workers and probability p in CrowdK.

monetary cost. Assuming that the answer of the workers is always correct, Max only optimized the number of questions under the latency constraint.

Specifically, when $k = 1$, Max was implemented by ConstantSequence in our problem setting. When $k > 1$, Max was iteratively performed to identify the top-1 result k times. Because this extension incurs excessive latency time, it violates the constraint for latency. As indicated in Fig. 11(a) and (b), CrowdK has a less number of questions than Max, because the tree building step ($k = 1$) in CrowdK can minimize the number of questions. As k increases, the gap between Max and CrowdK widens. For example, when $k = 20$, CrowdK has seven times less questions than Max. This is because CrowdK updates the tree selectively by using pre-obtained precedence between items; Max does not address the tree updating step.

Accuracy: We first validate the accuracy of top-k answers using a constant error model. Fig. 12 depicts the accuracy of top-k answers with varying ω and p in CrowdK. (Similar trends were also observed using other algorithms as indicated in Fig. 13.) Regardless of n and τ, the accuracy of the top-k answers increases with the number ω of workers per question and the probability p of correct answers. That is, we can improve the accuracy of the top-k answers by adjusting either ω or p.

Furthermore, it is also found that the accuracy of the top-k answers is similar in all algorithms when ω and p are fixed (Fig. 13). Although the accuracies of the top-k answers are marginally different, they indicate similar trends with varying ω and p. This is because the different number of questions for each algorithm does not significantly affect the accuracy of the top-k answers. In particular, because CrowdK generates questions selectively to determine the top-k answers by ignoring unnecessary questions, the accuracy of CrowdK is similar to the other algorithms. Because CrowdK generates the least number of answers with similar accuracy, it can be thus considered the overall winner.

We also evaluate the accuracy of top-k answers using different assignment methods (Fig. 14). Specifically, CrowdK (Static) is the conventional assignment method using a constant error model in CrowdK; CrowdK (Dynamic) is a heuristic assignment method in CrowdK, where the number of workers is dynamically assigned depending on the importance of the questions (Section 4.4).

It is consistently found that CrowdK (Dynamic) shows higher accuracy than CrowdK (Static) for all parameter settings of ω and p. That means that the assignment of CrowdK (Dynamic) is effective for improving the accuracy of the top-k answers by distinguishing the importance of the questions. When $\omega = 3$ or $p = 0.9$ is given, the improvement gap is maximized.

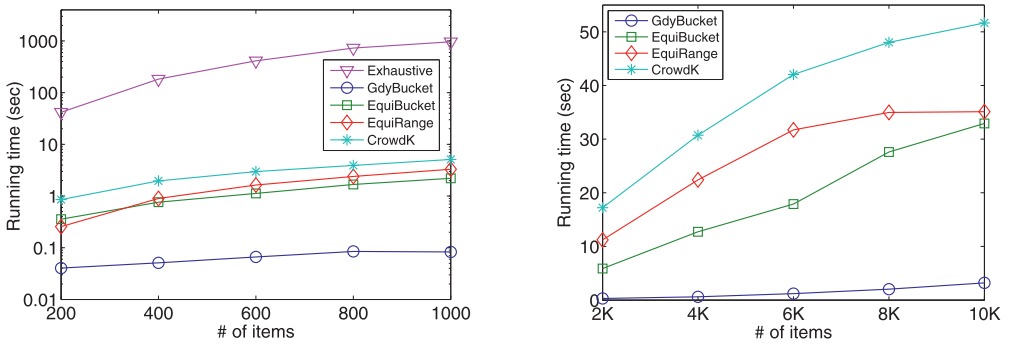


Fig. 15. Running time of five algorithms in small- and large-scale datasets.

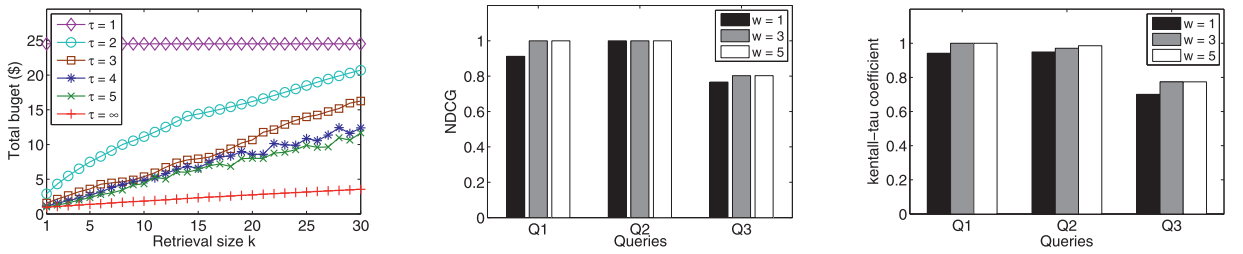


Fig. 16. Monetary cost and the accuracy using MTurk in real-life datasets ($n = 50$).

Running time: Fig. 15(a) compares the running time of the algorithms in the small-scale datasets. Note that the y-axis is log-scaled. The gap between Exhaustive and the other algorithms increases as the number of items increases. This is because the number of possible parameter combinations in Exhaustive increases exponentially, while the search space of the other algorithms increases linearly. Fig. 15(b) evaluates the proposed four algorithms in large-scale datasets.

Because building a tree in GdyBucket does not affect the number of questions, it indicates a constant performance. However, EquiBucket and EquiRange are linearly slower as the number of questions increases. GdyBucket is the fastest, and EquiBucket is marginally faster than EquiRange. CrowdK shows the slowest running time. The trade-off between the running time and the number of questions is consistent with the analysis in Table 3.

5.2. Evaluation with real-life datasets

We evaluate CrowdK with Amazon Mechanical Turk (AMT), a well-known crowdsourcing platform. We collect four real-life datasets: (1) *Squares* contain 50 images with the same color whose sizes are $\{(30+5i)\times(30+5i)|i\in[0,50)\}$, (2) *Rectangles* contain 50 images with different colors whose sizes are $\{(30+3i)\times(40+5i)|i\in[0,50)\}$, (3) *Animals* include 50 images ranging from bacteria to whale, and (4) *Statue of Liberty* includes 20 images, all related to the "Statue of Liberty". Fig. 17 represents a set of the 20 images for "Statue of Liberty". They are collected with the Google image search engine.

The set of 50 animals includes {*ant, armadillo, bacteria, bat, bear, beaver, bee, beetle, butterfly, camel, cat, chameleon, cheetah, chicken, chipmunk, dolphin, dugong, elephant, fox, giraffe, gorilla, hippo, horse, komododragon, leopard, lion, lobster, louse, manatee, meerkat, mosquito, moth, mouse, octopus, ostrich, owl, polarbear, praying mantis, rhinoceros, scorpion, shark, snake, spider, starfish, tiger, toad, walrus, whale, wolf, worm*}. It is used for evaluating adult size and the degree of danger of the animals.

We stress that the number n of items in real-life datasets can be set arbitrarily large as indicated in synthetic datasets. However, as we have already experimented extensively with large n in Section 5.1, we used a modest size n in the real-life experiments. Because CrowdK shows the best performance for monetary cost in Section 5.1, CrowdK is primarily evaluated in the real-life experiments. Using the four real-life datasets, the following five top-k queries are performed:

Q1: Find top-k squares by *area*
Q2: Find top-k rectangles by *area*
Q3: Find top-k animals by *adult size*
Q4: Find top-k animals by *degree of danger*
Q5: Find top-k images by *relevance* to 'Statue of Liberty'

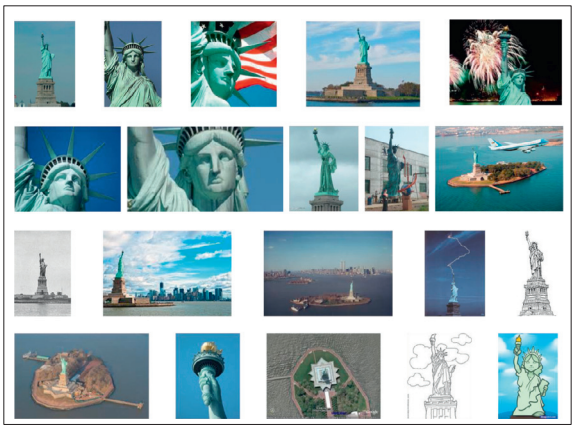


Fig. 17. Photos of "Statue of Liberty".

Table 5
HIT statistics per top-k query ($\tau = 4$).

	Q1	*Q2*	*Q3*	*Q4*	*Q5*
# of items	50	50	50	50	20
Retrieval size	5	5	5	10	20
Est. # of questions	128	128	128	214	116
Est. cost	$2.6	$2.6	$2.6	$4.3	$2.4
Avg. time per HIT (s)	19	22	32	26	22
Latency (min)	65	97	106	39	25

By design, *Q1*–*Q3* have ground truths; *Q4* and *Q5* do not. For *Q1* and *Q2*, we could intuitively obtain the ground truth by calculating areas. For *Q3*, we collect the ground truth for the top-5 answers based on animal information in Wikipedia. For *Q4* and *Q5*, because there is no ground-truth, we empirically observed if the top-k answers are valid. From *Q1* to *Q5*, it is increasingly more ambiguous and challenging to crowds. By comparing the top-k answers with ground truth in *Q1*–*Q3*, we measured the accuracy of the top-k answers by varying the difficulty of tasks. For these top-k queries, machine-based algorithms have difficulty in obtaining correct answers; however, crowds can intuitively recognize correct answers. Further, we quantified the accuracy of all the independent pair-wise answers returned by the workers, which were measured by the normalized *Kendall-tau* coefficient. This is useful for measuring the agreement of the answers. A score closer to one indicates that the answers of the workers are more correct.

To validate top-k queries in real-life datasets from AMT, we use only human-based computation. (To further reduce monetary cost, it is possible to combine machine- and human- computation as discussed in Section 1.) For each dataset, we varied τ, k, and ω. When the parameters are given, CrowdK identifies the values of the buckets and ranges. In this process, we generate tasks, called *Human Intelligence Tasks (HITs)*. Given $|\mathcal{Q}_i|$ questions at ith round, one HIT includes five pair-wise questions. For each HIT, ω workers are assigned and monetary cost δ per HIT is set to \$0.02. When $\omega = 5$, $C(A, \mathcal{E})$ is calculated as: $0.02 \times 5 \times \sum_{i=1}^{\tau} \lceil \frac{|\mathcal{Q}_i|}{5} \rceil$. To reduce spam workers, we used two qualification conditions embedded in AMT for workers: (1) HIT approval rate $\geq$ 95%, and (2) # of approved HIT $\geq$ 100. To manage the latency, we assume that each round has 30 min. Based on these assumptions, the response time for the top-k queries is calculated as $\#\ of\ \ rounds \times 30\ \ minutes$. Note that the response time per round can be different depending on the query.

Monetary cost: Table 5 reports the statistics for the top-k queries in AMT. For each query, we set $\tau = 4$ and $k = 5$ for *Q1*–*Q3*, $k = 10$ for *Q4*, and $k = 20$ for *Q5*. If all pair-wise questions are asked by the one-shot strategy, it requires \$24.5 for 50 items. However, we can save more than \$20 by performing an iterative strategy ($\tau = 4$). From *Q1* to *Q3*, it is observed that all answers are collected in 30 min per round. The response time for Q1 and Q2 are significantly less than the estimated response time, *i.e.*, 120 min. Further, the average time per HIT and latency tends to increase. Because the questions in Q3 are more difficult than those in Q1, the crowd requires additional time to find answers for Q3.

Fig. 16(a) depicts the costs over varying τ and k in CrowdK. As the retrieval size increases, the cost of CrowdK increases as well. When $\tau = 4$ and $k \leq 10$ are given, the saving cost is approximately \$20, compared to the one-shot strategy ($\tau = 1$). Most notably, when $\tau = 5$, CrowdK consistently saves more than \$10 over all retrieval sizes ($1 \leq k \leq 30$).

Accuracy: Fig. 16(b) reports NDCG for the three queries *Q1*–*Q3*. For *Q1* and *Q2*, the top-5 answers are all correct when $w \geq 3$. For *Q3*, our top-5 answers are {*whale, elephant, giraffe, hippo, rhinocerous*}, whereas the ground truth is {*whale, shark, dolphin, elephant, giraffe, hippo, rhinocerous, ...*}. Fig. 16(c) reports the Kendall-tau coefficient to quantify the agreement of the answers for all pair-wise questions. When $\omega = 5$, the Kendall-tau coefficient is more than 0.75 for all three queries. That is, when majority voting is used with $\omega = 5$, the workers identifies the correct answers for approximately 70% of the questions.

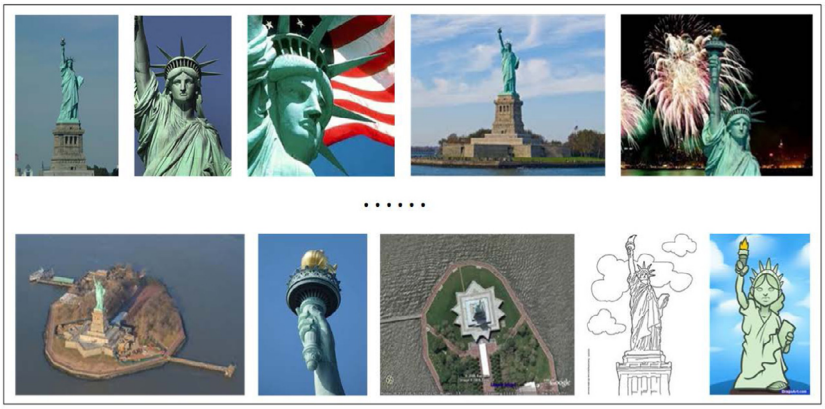


Fig. 18. Top-5 and bottom-5 answers for Q5.

Finally, we validate the top-10 and top-20 answers for two subjective queries, *Q4* and *Q5*. For *Q4*, the top-10 answers by the crowd are {*tiger, lion, bacteria, hippo, shark, bear, leopard, polarbear, komododragon, rhinoceros*} including carnivores and microorganisms. For *Q5*, Fig. 18 displays the both top-5 and bottom-5 answers by the crowd. To evaluate *Q5*, all images are considerably related to the "Statue of Liberty". Therefore, it is rather fuzzy and ambiguous even to humans. Although there is no ground truth for *Q5*, we believe that the top-5 answers collected from the crowd are acceptable.

6. Related work

Crowdsourcing has been actively used to perform intelligent tasks with large-scale human computation. It was first used to collect human judgments for character recognition in the ESP game [40]. Later, it was applied to various research areas such as information retrieval, natural language processing, machine learning, and human-computer interaction. It can enhance the accuracy of results in answering subjective search queries [6], understanding topics in microblogs [19], image tagging [43], editing documents as natural language understanding [4], and choosing a representative frame to summarize a video [3]. Further, it can be combined with active learning to identify reliable samples [31]. It has also been used for evaluating human-computer interaction [9,36]. A detailed survey of crowdsourcing can be found in [13,44]. Recently, [18] discussed open challenges and issues addressing crowdsourcing systems.

In this paper, we studied crowd-enabled top-*k* queries to support effective data retrieval. We elaborate on several categories related to the proposed work.

Crowdsourcing systems and algorithms: Crowd-enabled database systems have been popular in the database community, *e.g.*, CrowdDB [16], Deco [33], Qurk [28], CDAS [26], CyLog/Crowd4U [30], and AskIt! [5]. These systems have extended existing database systems to acquire missing information and to perform cognitive comparisons by crowds. Parameswaran et al. [34] minimized the number of questions for finding a target node in a given graph, Gomes et al. [20] addressed crowd-enabled clustering, and Amsterdamer et al. [1] integrated data mining with crowdsourcing. Moreover, some existing work has extended machine-based operations into crowd-enabled operations, *e.g.*, selection [17,32,37], join [29,41,42], group by [12], and skyline queries [25,27].

*Top-*k *query algorithms:* Top-*k* queries have been widely used for retrieving the best *k* ordered items in many real applications. The existing works [14,23] mainly focused on minimizing the number of accessed items without a full data scan. While optimizing the efficiency in conventional databases, they do not employ human computation in that case that it is difficult for users to define an appropriate scoring function. Toward this goal, we formulate the crowd-enabled top-*k* queries by minimizing the monetary cost while latency is constrained.

Our work: The top-*k* query formulation is closely related to crowd-enabled max queries [21,38,39], sorting [29], top-*k* queries [10,12], and top-*k* set queries [35]. However, it clearly differs from them. Table 6 represents the comparisons between existing work and our work based on key crowdsourcing factors.

Specifically, [21] transformed max queries into the judgment problem in a graph and focused on resolving the conflicts of answers collected from crowds. Venetis et al. [38] addressed max queries by maximizing the accuracy of the query results while both the latency and monetary cost are constrained. Later, Verroios et al. [39] used the latency function as input to address max queries. The latency function is used to estimate the running time to return the answers for pair-wise comparisons. Because [38,39] addressed max queries, they did not consider an updating process for top-*k* queries. (The time complexity of max queries in [38,39] is equal to building the tournaments in Table 3.) They are thus limited in their ability to expand into top-*k* queries. It is possible to iterate max queries *k* times to support top-*k* queries.

Marcus et al. [29] addressed a sorting operation by optimizing monetary cost; however, it did not impose other constraints. Further, Davidson et al. [12] addressed crowd-enabled top-*k* queries by theoretically minimizing monetary cost under a constraint for quality of answers. To analyze the error of the answers, Davidson et al. [12] proposed a *variable er-*

Table 6
Comparison between existing work and ours.

	Type	Latency	Cost	Quality
Guo et al. [21]	max queries		✓	✓
Venetis et al. [38]	max queries	✓	✓	✓
Verroios et al. [39]	max queries	✓	✓	✓
Marcus et al. [29]	sorting		✓	
Davidson et al. [12]	top-k queries		✓	✓
Ciceri et al. [10]	top-k queries		✓	✓
Polychronopoulos et al. [35]	top-k set		✓	✓
Ours	top-k queries	✓	✓	✓

ror model by extending the existing *constant error model* [15], neglecting the constraint for latency. Polychronopoulos et al. [35] adopted a *top*-k *set* where the precedence among top-k items was not considered and ignored the constraint for latency. Recently, Ciceri et al. [10] proposed a crowd-enabled top-k query algorithm in a probabilistic data setting. Because assuming that the probabilities for preferences between items are given, Ciceri et al. [10] only focused on selecting more useful questions under the cost constraint. Arguably, the proposed top-k query formulation is more difficult than [35], more generalized than [10,12,21,38,39], and more practical than [12,29] in on-line environments.

7. Discussion

In this section, we briefly discuss possible extensions of the proposed method and interesting future work.

Using multiple-choice questions: Although we mainly assume pair-wise questions, it is possible to apply multiple choice questions with minor modification. First, estimating the number of questions should be modified. If the number of items in a bucket is less than or equal to m items, the number of questions for multiple choices can be reduced to $\lceil \frac{b(b-1)}{m(m-1)} \rceil$ as discussed in [29]. Further, computing the number of workers to manage the accuracy of the top-k answers should be modified. One possible solution is to exploit the error model for multiple choices discussed in [38]. It is open to adopt a new model to estimate the number of questions and to manage the accuracy of answers for multiple choice questions.

Updating retrieval sizes incrementally: Once a tree is created, it can be reused to obtain additional top-k answers in an interactive manner. In this case, because there is no constraint for latency, additional top-k answers are iteratively identified by updating the tree.

Integrating pre-obtained knowledge: If the precedences between items are partially pre-defined without using crowdsourcing, they can be represented by a tree or a forest, where multiple root nodes can exist. When building and updating a tree, CrowdK can thus remove unnecessary questions using the pre-defined knowledge as discussed in Section 4.2.

Formulating other related problems: We focused on minimizing monetary cost while latency is constrained. However, the proposed parameterized framework can also be utilized to address other related problems. For example, when only the quality of answers is constrained, it can be leveraged to find a set of possible execution plans with a trade-off between latency and monetary cost. The execution plans can be represented as a *pareto-optimal* set (or a *skyline*).

8. Conclusion

In this paper, we studied the problem of answering top-k queries with crowds. This was formulated to minimize the number of questions under the constraint of the number of rounds. To address this problem, we modeled a parameterized framework with two parameters $\mathcal{B}$ and $\mathcal{R}$. Based on the framework, we developed three methods: greedy, equi-sized, and dynamic programming, and proposed four algorithms: GdyBucket, EquiBucket, EquiRange, and CrowdK, by combining the three methods. Lastly, we empirically evaluated the trade-off between our proposals with respect to monetary cost, accuracy of the top-k answers, and running time. We found that CrowdK required 20 times less than other algorithms in terms of the monetary cost without sacrificing the quality of the answers.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2015R1C1A1A01055442).

Appendix A. Mathematical proofs

We prove the following lemmas and theorems used in this paper.

Theorem 1 (Horizontal optimization)**.** *When an item set $\mathcal{E}$ is partitioned into m buckets, $\mathcal{B}_i = \langle b_{i1}, \dots b_{im} \rangle$ minimizes $|\mathcal{Q}_i|$ if $b_{i*}^{min} = \lfloor |\mathcal{E}|/m \rfloor$ and $b_{i*}^{max} = \lceil |\mathcal{E}|/m \rceil$ hold.*

Proof. Given $\mathcal{B}_i = \langle b_{i1}, \dots b_{im} \rangle$, the number $|\mathcal{Q}_i|$ of questions is calculated as $\sum_{j=1}^{m} f(b_{ij})$ such that $f(b_{ij}) = \binom{b_{ij}}{2} = \frac{b_{ij}(b_{ij}-1)}{2}$. Because $f(b_{ij})$ is the convex function, it increases quadratically with b_{ij}.

Based on the property of the convex function, we prove this by contradiction. Assume that $\mathcal{B}'_i = \langle b_{i1} + \delta, b_{i2} - \delta, b_{i3}, \cdots, b_{im} \rangle$ such that $1 \le \delta \le b_{i*}^{min} - 1$ minimizes $|\mathcal{Q}_i|$. Given an arbitrary bucket size b, $\binom{b}{2} = \binom{b-1}{2} + (b-1)$ holds by the formula of the binomial coefficient. In this case, $\binom{b}{2} - \binom{b-\delta}{2} = \sum_{i=1}^{\delta} b - i$ holds.

Using this equation, the following inequality between $\mathcal{B}_i$ and $\mathcal{B}'_i$ also holds: $\binom{b}{2} + \binom{b}{2} < \binom{b-\delta}{2} + \binom{b+\delta}{2}$, because $\sum_{i=1}^{\delta} b - i < \sum_{i=0}^{\delta-1} b + i$. For another case such that $b_{i1} = b_{i*}^{min}$ and $b_{i2} = b_{i*}^{max}$, we can also demonstrate the same inequality between $\mathcal{B}_i$ and $\mathcal{B}'_i$. Because of the convex property, it is also trivial to demonstrate that $\mathcal{B}'_i$ with different values cannot minimize the number of questions. Consequently, $\mathcal{B}'_i$ cannot minimize $|\mathcal{Q}_i|$, which is a contradiction. □

Lemma 1. *For* $2 \le i \le \tau_B$ *and* $2^{i-1} + 1 \le j \le n$, $V[i, j] = \min_{b \in [2, j-1]} V[i-1, \lceil \frac{j}{b} \rceil] + \mathcal{Q}(j, \lceil \frac{j}{b} \rceil)$.

Proof. Let V denote a τ_B-by-n matrix $V[1, \dots, \tau_B, 2, \dots, n]$. For $1 \le i \le \tau_B$ and $2 \le j \le n$, $V[i, j]$ stores the minimum number of questions for $\tau_B = i$ rounds and $|\mathcal{E}| = j$. We prove that the recursive equation minimizes the number of questions by induction:

1. Base case ($i = 1$): As the initial step, $V[1, j]$ is set by $\binom{j}{2}$ for $2 \le j \le n$. This setting minimizes the number of questions.
2. Hypothesis ($i = m$): By inductive hypothesis, for $2^{i-1} + 1 \le j \le n$, $V[i, j]$ minimizes the number of questions.
3. Induction ($i = m + 1$): For $2^{i-1} + 1 \le j \le n$, the possible range of bucket size b is $[2, j-1]$. Given $|\mathcal{E}_i| = j$ and b, the number of questions is computed by $\mathcal{Q}(j, \lceil \frac{j}{b} \rceil)$, and $|\mathcal{E}_{i+1}|$ is determined by $\lceil \frac{j}{b} \rceil$. Given $|\mathcal{E}_{i+1}| = \lceil \frac{j}{b} \rceil$, $V[i-1, \lceil \frac{j}{b} \rceil]$ minimizes the number of questions. Therefore, when $V[i-1, \lceil \frac{j}{b} \rceil]$ is feasible, the following recursive equation minimizes the number of questions.

$$V[i, j] = \min_{2 \le b \le j-1} V[i-1, \lceil \frac{j}{b} \rceil] + \mathcal{Q}(j, \lceil \frac{j}{b} \rceil).$$

By induction, this recursive equation minimizes the number of questions for $2 \le i \le \tau_B$ and $2^{i-1} + 1 \le j \le n$. □

Theorem 2 (Vertical optimization). *Given* n *and* τ_B, $\widetilde{\mathcal{B}}$ *minimizes* $|\mathcal{Q}_B|$, *if* $\widetilde{\mathcal{B}}$ *is determined by* $B[\tau_B, n]$.

Proof. To prove this, we define the following recursive equation for $V[\tau_B, n]$:

- As the initial setting, $V[i, j]$ is set as:
  - When $i = 1$, $V[1, j] = \binom{j}{2}$ for $2 \le j \le n$.
  - When $2 \le i \le \tau_B$, $V[i, j] = \infty$ for $2 \le j < 2^{i-1}$, which means infeasible cases.
- As the recursive setting, for $2 \le i \le \tau_B$ and $2^{i-1} + 1 \le j \le n$, $V[i, j]$ is set by the following equation:

$$V[i, j] = \min_{2 \le b \le j-1} V[i-1, \lceil \frac{j}{b} \rceil] + \mathcal{Q}(j, \lceil \frac{j}{b} \rceil).$$

Because this recursive equation satisfies the principle of optimality by Lemma 1, $V[\tau_B, n]$ minimizes the number of questions. Because B corresponds to V, $B[\tau_B, n]$ is a sequence of vertical bucket sizes that minimizes the number of questions. □

Lemma 2. *When* r_i *and* $\mathcal{T}$ *are given,* (e_i, e_j) *can be skipped, if (1)* $e_i \in$ *Ancestor*(e_j), *(2)* $e_j \in$ *Ancestor*(e_i), *or (3)* $|$*Ancestor*$(e_i) \cup$ *Ancestor*$(e_j)| \ge r_i$.

Proof. Let *Ancestor*(e_i) denote a set of ancestors of e_i in $\mathcal{T}$. For the first and second case, because (e_i, e_j) can be derived by transitivity, it does not affect determining the top-r_i answers. For the third case, when $|$*Ancestor*$(e_i) \cup$ *Ancestor*$(e_j)| \ge r_i$, both e_i and e_j cannot be in the top-u_i answers. That is, if e_i is a top-u_i candidate, e_j is not a top-u_i candidate. If e_j is a top-u_i candidate, e_i is not a top-u_i candidate. Therefore, (e_i, e_j) does not affect determining the top-u_i answers. □

References

[1] Y. Amsterdamer, Y. Grossman, T. Milo, P. Senellart, Crowd mining, in: Proceedings of International Conference on Management of Data (SIGMOD), 2013, pp. 241–252.

[2] D.W. Barowy, C. Curtsinger, E.D. Berger, A. McGregor, Automan: A Platform for Integrating Human-Based and Digital Computation, Commun. ACM 59 (6) (2016) 102–109.

[3] M.S. Bernstein, J. Brandt, R.C. Miller, D.R. Karger, Crowds in two seconds: enabling realtime crowd-powered interfaces, in: Proceedings of ACM Symposium on User Interface Software and Technology (UIST), 2011, pp. 33–42.

[4] M.S. Bernstein, G. Little, R.C. Miller, B. Hartmann, M.S. Ackerman, D.R. Karger, D. Crowell, K. Panovich, Soylent: a word processor with a crowd inside, in: Proceedings of ACM Symposium on User Interface Software and Technology (UIST), 2010, pp. 313–322.

[5] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, W.C. Tan, Asking the right questions in crowd data sourcing, in: Proceedings of International Conference on Data Engineering (ICDE), 2012, pp. 1261–1264.

[6] A. Bozzon, M. Brambilla, S. Ceri, Answering search queries with crowdsearcher, in: Proceedings of International World Wide Web Conferences (WWW), 2012, pp. 1009–1018.

[7] G. Brightwell, P. Winkler, Counting linear extensions is #p-complete, in: Proceedings of Symposium on the Theory of Computing (STOC), 1991, pp. 175–181.

[8] R. Bubley, M.E. Dyer, Faster random generation of linear extensions, in: Proceedings of Symposium on Discrete Algorithms (SODA), 1998, pp. 350–354.

[9] L.B. Chilton, G. Little, D. Edge, D.S. Weld, J.A. Landay, Cascade: crowdsourcing taxonomy creation, in: Proceedings of ACM Special Interest Group on Computer–Human Interaction (SIGCHI), 2013, pp. 1999–2008.
[10] E. Ciceri, P. Fraternali, D. Martinenghi, M. Tagliasacchi, Crowdsourcing for top-*k* query processing over uncertain data, IEEE Trans. Knowl. Data Eng. 28 (1) (2016) 41–53.
[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009.
[12] S.B. Davidson, S. Khanna, T. Milo, S. Roy, Using the crowd for top-*k* and group-by queries, in: Proceedings of International Conference on Database Theory (ICDT), 2013.
[13] A. Doan, R. Ramakrishnan, A.Y. Halevy, Crowdsourcing systems on the world-wide web, Commun. ACM 54 (4) (2011) 86–96.
[14] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, J. Comput. Syst. Sci. 66 (4) (2003) 614–656.
[15] U. Feige, P. Raghavan, D. Peleg, E. Upfal, Computing with noisy information, SIAM J. Comput. 23 (5) (1994) 1001–1018.
[16] M.J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, R. Xin, CrowdDB: answering queries with crowdsourcing, in: Proceedings of International Conference on Management of Data (SIGMOD), 2011, pp. 61–72.
[17] J. Gao, X. Liu, B.C. Ooi, H. Wang, G. Chen, An online cost sensitive decision-making method in crowdsourcing systems, in: Proceedings of International Conference on Management of Data (SIGMOD), 2013, pp. 217–228.
[18] H. Garcia-Molina, M. Joglekar, A. Marcus, A.G. Parameswaran, V. Verroios, Challenges in data crowdsourcing, IEEE Trans. Knowl. Data Eng. 28 (4) (2016) 901–911.
[19] S. Ghosh, N.K. Sharma, F. Benevenuto, N. Ganguly, P.K. Gummadi, Cognos: crowdsourcing search for topic experts in microblogs, in: Proceedings of International Conference on Research and Development in Information Retrieval (SIGIR), 2012, pp. 575–590.
[20] R. Gomes, P. Welinder, A. Krause, P. Perona, Crowdclustering, in: Proceedings of Neural Information Processing Systems (NIPS), 2011, pp. 558–566.
[21] S. Guo, A.G. Parameswaran, H. Garcia-Molina, So who won?: Dynamic max discovery with the crowd, in: Proceedings of International Conference on Management of Data (SIGMOD), 2012, pp. 385–396.
[22] D. Haas, J. Wang, E. Wu, M.J. Franklin, Clamshell: Speeding up crowds for low-latency data labeling, Proc. VLDB Endow. 9 (4) (2015) 372–383.
[23] I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-*k* query processing techniques in relational database systems, ACM Comput. Surv. 40 (4) (2008).
[24] Y. Kim, K. Shim, Parallel top-*k* similarity join algorithms using MapReduce, in: Proceedings of International Conference on Data Engineering (ICDE), 2012, pp. 510–521.
[25] J. Lee, D. Lee, S. Kim, CrowdSky: crowdsourced skyline query processing, in: Proceedings of International Conference on Extending Database Technology (EDBT), 2016.
[26] X. Liu, M. Lu, B.C. Ooi, Y. Shen, S. Wu, M. Zhang, CDAS: a crowdsourcing data analytics system, Proc. VLDB Endow. 5 (10) (2012) 1040–1051.
[27] C. Lofi, K.E. Maarry, W.-T. Balke, Skyline queries in crowd-enabled databases, in: Proceedings of International Conference on Extending Database Technology (EDBT), 2013.
[28] A. Marcus, E. Wu, D.R. Karger, S. Madden, R.C. Miller, Demonstration of Qurk: a query processor for human operators, in: Proceedings of International Conference on Management of Data (SIGMOD), 2011, pp. 1315–1318.
[29] A. Marcus, E. Wu, D.R. Karger, S. Madden, R.C. Miller, Human-powered sorts and joins, Proc. VLDB Endow. 5 (1) (2011) 13–24.
[30] A. Morishima, N. Shinagawa, T. Mitsuishi, H. Aoki, S. Fukusumi, CyLog/Crowd4U: a declarative platform for complex data-centric crowdsourcing, Proc. VLDB Endow. 5 (12) (2012) 1918–1921.
[31] B. Mozafari, P. Sarkar, M.J. Franklin, M.I. Jordan, S. Madden, Scaling up crowd-sourcing to very large datasets: a case for active learning, Proc. VLDB Endow. 8 (2) (2014) 125–136.
[32] A.G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, J. Widom, Crowdscreen: algorithms for filtering data with humans, in: Proceedings of International Conference on Management of Data (SIGMOD), 2012, pp. 361–372.
[33] A.G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, J. Widom, Deco: declarative crowdsourcing, in: Proceedings of International Conference on Information and Knowledge Management (CIKM), 2012, pp. 1203–1212.
[34] A.G. Parameswaran, A.D. Sarma, H. Garcia-Molina, N. Polyzotis, J. Widom, Human-assisted graph search: it's okay to ask questions, Proc. VLDB Endow. 4 (5) (2011) 267–278.
[35] V. Polychronopoulos, L. de Alfaro, J. Davis, H. Garcia-Molina, N. Polyzotis, Human-powered top-*k* lists, in: Proceedings of International Workshop on the Web and Databases (WebDB), 2013.
[36] A.J. Quinn, B.B. Bederson, Human computation: a survey and taxonomy of a growing field, in: Proceedings of ACM Special Interest Group on Computer-Human Interaction (SIGCHI), 2011, pp. 1403–1412.
[37] B. Trushkowsky, T. Kraska, M.J. Franklin, P. Sarkar, Crowdsourced enumeration queries, in: Proceedings of International Conference on Data Engineering (ICDE), 2013, pp. 673–684.
[38] P. Venetis, H. Garcia-Molina, K. Huang, N. Polyzotis, Max algorithms in crowdsourcing environments, in: Proceedings of International World Wide Web Conferences (WWW), 2012, pp. 989–998.
[39] V. Verroios, P. Lofgren, H. Garcia-Molina, tDP: An optimal-latency budget allocation strategy for crowdsourced MAXIMUM operations, in: Proceedings of International Conference on Management of Data (SIGMOD), 2015, pp. 1047–1062.
[40] L. von Ahn, L. Dabbish, Designing games with a purpose, Commun. ACM 51 (8) (2008) 58–67.
[41] J. Wang, T. Kraska, M.J. Franklin, J. Feng, CrowdER: crowdsourcing entity resolution, Proc. VLDB Endow. 5 (11) (2012) 1483–1494.
[42] J. Wang, G. Li, T. Kraska, M.J. Franklin, J. Feng, Leveraging transitive relations for crowdsourced joins, in: Proceedings of International Conference on Management of Data (SIGMOD), 2013, pp. 229–240.
[43] T. Yan, V. Kumar, D. Ganesan, CrowdSearch: exploiting crowds for accurate real-time image search on mobile phones, in: Proceedings of International Conference on Mobile Systems, Applications, and Services (MobiSys), 2010, pp. 77–90.
[44] M.-C. Yuen, I. King, K.-S. Leung, A survey of crowdsourcing systems, in: Proceedings of International Conference on Social Computing (SocialCom), 2011, pp. 766–773.